

THOMSON  
COURSE TECHNOLOGY

Professional • Trade • Reference

# PROGRAMMING ROLE PLAYING GAMES WITH DIRECTX®

2<sup>ND</sup> EDITION

JIM ADAMS



Premier  
  
Press

*Джим Адамс*

# **ПРОГРАММИРОВАНИЕ РОЛЕВЫХ ИГР С DirectX**

*2-е издание*

Джим Адамс

## Программирование ролевых игр с DirectX (2-е издание)

Thomson Course Technology PTR, 2004

ISBN: 1-59200-315-X

### Аннотация

Ни один из жанров игр не способствует погружению игрока в свой мир так, как это делают ролевые игры. В них игрок прорывается сквозь барьер существующий между его действиями и фантастическим миром и становится жителем того мира. Приготовьтесь воплотить свои идеи в жизнь и создать собственную ролевую игру! Книга «Программирование ролевых игр с DirectX» покажет вам как создать графическую библиотеку и механизм сражений, как управлять игроками, использовать скрипты и предметы, и как сделать вашу игру многопользовательской. Книга разделяет ролевую игру на базовые составляющие, подробно исследует их и показывает, как вы можете применять их в своем игровом проекте. Вы научитесь рисовать с DirectX Graphics, воспроизводить звуки и музыку с DirectX Audio, работать с сетью с DirectPlay и взаимодействовать с игроком с DirectInput. Когда вы закончите чтение книги, у вас будет достаточно знаний, чтобы создать собственную законченную ролевую игру.

Понравился перевод? Не забудьте покормить переводчика.

**WebMoney: Z165309743603, R229843962424**

# Оглавление

<b>БЛАГОДАРНОСТИ .....</b>	<b>19</b>
<b>ОБ АВТОРЕ.....</b>	<b>21</b>
<b>ВВЕДЕНИЕ .....</b>	<b>23</b>
О ЧЕМ ЭТА КНИГА.....	23
ДЛЯ КОГО ПРЕДНАЗНАЧЕНА КНИГА .....	24
ОРГАНИЗАЦИЯ КНИГИ .....	24
СОДЕРЖИМОЕ CD-ROM .....	25
ТИПОГРАФСКИЕ СОГЛАШЕНИЯ.....	25
ЧТО НУЖНО ДЛЯ НАЧАЛА.....	26
<b>ЧАСТЬ I РАБОТА С КНИГОЙ .....</b>	<b>27</b>
<b>ГЛАВА 1 ПОДГОТОВКА К РАБОТЕ С КНИГОЙ.....</b>	<b>29</b>
DIRECTX.....	29
Подготовка DirectX .....	31
Отладочные и дистрибутивные версии .....	31
Установка DirectX 9.....	31
Установка DirectMusic Producer .....	33
Включаемые файлы и библиотеки .....	33
ПОДГОТОВКА КОМПИЛЯТОРА .....	33
УКАЗАНИЕ КАТАЛОГОВ ДЛЯ DirectX .....	34
СВЯЗЫВАНИЕ С БИБЛИОТЕКАМИ.....	36
УСТАНОВКА ПОВЕДЕНИЯ ПО УМОЛЧАНИЮ ДЛЯ CHAR.....	39
Окончательные и отладочные версии .....	40
Многопоточные библиотеки.....	40
ПРОГРАММИРОВАНИЕ ДЛЯ WINDOWS .....	41
Потоки и многопоточность .....	41
Критические секции .....	43
ИСПОЛЬЗОВАНИЕ COM .....	44
Инициализация COM .....	45
IUnknown .....	45
Инициализация и освобождение объектов.....	46
Запрос интерфейсов.....	47
ПОТОК ВЫПОЛНЕНИЯ ПРОГРАММЫ.....	48



<b>МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ .....</b>	<b>49</b>
<b>СОСТОЯНИЯ И ПРОЦЕССЫ.....</b>	<b>51</b>
Состояния приложения .....	51
ПРОЦЕССЫ .....	55
<b>ОБРАБОТКА ДАННЫХ ПРИЛОЖЕНИЯ.....</b>	<b>57</b>
ИСПОЛЬЗОВАНИЕ УПАКОВКИ ДАННЫХ .....	57
ТЕСТИРОВАНИЕ СИСТЕМЫ УПАКОВКИ ДАННЫХ .....	58
<b>ПОСТРОЕНИЕ КАРКАСА ПРИЛОЖЕНИЯ .....</b>	<b>60</b>
<b>СТРУКТУРИРОВАНИЕ ПРОЕКТА .....</b>	<b>65</b>
<b>ЗАВЕРШАЕМ ПОДГОТОВКУ .....</b>	<b>65</b>
 <b>ЧАСТЬ II ОСНОВЫ DIRECTX.....</b>	 <b>67</b>
 <b>ГЛАВА 2 РИСОВАНИЕ С DIRECTX GRAPHICS .....</b>	 <b>69</b>
<b>СЕРДЦЕ ТРЕХМЕРНОЙ ГРАФИКИ .....</b>	<b>70</b>
СИСТЕМЫ КООРДИНАТ .....	71
КОНСТРУИРОВАНИЕ ОБЪЕКТОВ .....	73
СПИСКИ, ПОЛОСЫ И ВЕЕРА .....	74
ПОРЯДОК ВЕРШИН .....	75
ОКРАШИВАНИЕ ПОЛИГОНОВ .....	76
ПРЕОБРАЗОВАНИЯ .....	76
<b>НАЧИНАЕМ РАБОТАТЬ С DIRECTX GRAPHICS .....</b>	<b>77</b>
КОМПОНЕНТЫ DIRECT3D .....	78
ИНИЦИАЛИЗАЦИЯ СИСТЕМЫ .....	79
ПОЛУЧЕНИЕ ИНТЕРФЕЙСА DIRECT3D .....	79
ВЫБОР ВИДЕОРЕЖИМА .....	79
УСТАНОВКА МЕТОДА ПОКАЗА .....	82
СОЗДАНИЕ ИНТЕРФЕЙСА УСТРОЙСТВА И ИНИЦИАЛИЗАЦИЯ ДИСПЛЕЯ .....	84
ПОТЕРЯ УСТРОЙСТВА .....	85
ЗНАКОМСТВО С D3DX .....	85
<b>ТРЕХМЕРНАЯ МАТЕМАТИКА .....</b>	<b>86</b>
МАТРИЧНАЯ АЛГЕБРА.....	86
КОНСТРУИРОВАНИЕ МАТРИЦ.....	87
КОМБИНИРОВАНИЕ МАТРИЦ.....	89
ПЕРЕХОД ОТ ЛОКАЛЬНЫХ КООРДИНАТ К КООРДИНАТАМ ВИДА.....	90
<b>ПЕРЕХОДИМ К РИСОВАНИЮ .....</b>	<b>92</b>
ИСПОЛЬЗОВАНИЕ ВЕРШИН .....	92
НАСТРАИВАЕМЫЙ ФОРМАТ ВЕРШИН .....	92
ИСПОЛЬЗОВАНИЕ БУФЕРА ВЕРШИН.....	94
Создание буфера вершин.....	95
Блокировка буфера вершин .....	96
Заполнение данных вершин.....	97
ПОТОК ВЕРШИН .....	98
ВЕРШИННЫЕ ШЕЙДЕРЫ .....	99
ПРЕОБРАЗОВАНИЯ .....	99
МИРОВОЕ ПРЕОБРАЗОВАНИЕ .....	100
ПРЕОБРАЗОВАНИЕ ВИДА .....	101

ПРЕОБРАЗОВАНИЕ ПРОЕКЦИИ .....	103
МАТЕРИАЛЫ И ЦВЕТА .....	105
ОЧИСТКА ПОРТА ПРОСМОТРА .....	106
НАЧАЛО И ЗАВЕРШЕНИЕ СЦЕНЫ .....	107
ВИЗУАЛИЗАЦИЯ ПОЛИГОНОВ .....	107
ПОКАЗ СЦЕНЫ.....	109
<b>ИСПОЛЬЗОВАНИЕ НАЛОЖЕНИЯ ТЕКСТУР.....</b>	<b>109</b>
ИСПОЛЬЗОВАНИЕ НАЛОЖЕНИЯ ТЕКСТУР В DIRECT3D .....	111
ЗАГРУЗКА ТЕКСТУРЫ.....	112
УСТАНОВКА ТЕКСТУРЫ .....	113
ИСПОЛЬЗОВАНИЕ ВЫБОРОК .....	115
ВИЗУАЛИЗАЦИЯ ТЕКСТУРИРОВАННЫХ ОБЪЕКТОВ .....	116
<b>АЛЬФА-СМЕШИВАНИЕ .....</b>	<b>117</b>
ВКЛЮЧЕНИЕ АЛЬФА-СМЕШИВАНИЯ.....	118
РИСОВАНИЕ С АЛЬФА-СМЕШИВАНИЕМ .....	119
КОПИРОВАНИЕ С УЧЕТОМ ПРОЗРАЧНОСТИ И АЛЬФА-ПРОВЕРКА .....	119
ЗАГРУЗКА ТЕКСТУР С ЦВЕТОВЫМ КЛЮЧОМ .....	120
ВКЛЮЧЕНИЕ АЛЬФА-ПРОВЕРКИ .....	121
ПРИМЕР КОПИРОВАНИЯ С УЧЕТОМ ПРОЗРАЧНОСТИ.....	122
ОСВЕЩЕНИЕ .....	123
ИСПОЛЬЗОВАНИЕ ТОЧЕЧНОГО СВЕТА.....	126
ИСПОЛЬЗОВАНИЕ ЗОНАЛЬНОГО СВЕТА.....	126
ИСПОЛЬЗОВАНИЕ НАПРАВЛЕННОГО СВЕТА .....	128
ФОНОВЫЙ СВЕТ .....	129
УСТАНОВКА СВЕТА.....	129
ИСПОЛЬЗОВАНИЕ НОРМАЛЕЙ.....	129
ДА БУДЕТ СВЕТ! .....	132
<b>ИСПОЛЬЗОВАНИЕ ШРИФТОВ.....</b>	<b>132</b>
СОЗДАНИЕ ШРИФТА .....	133
РИСОВАНИЕ ТЕКСТА .....	135
<b>ЩИТЫ .....</b>	<b>136</b>
<b>ЧАСТИЦЫ .....</b>	<b>138</b>
<b>СОРТИРОВКА ПО ГЛУБИНЕ И Z-БУФЕРИЗАЦИЯ.....</b>	<b>141</b>
<b>РАБОТА С ПОРТОМ ПРОСМОТРА .....</b>	<b>143</b>
<b>РАБОТА С СЕТКАМИ.....</b>	<b>144</b>
X-ФАЙЛЫ.....	145
ФОРМАТ X-ФАЙЛОВ .....	145
Использование иерархии фреймов .....	146
СОЗДАНИЕ X-ФАЙЛОВ С СЕТКАМИ.....	148
РАЗБОР X-ФАЙЛОВ.....	148
<b>СЕТКИ В БИБЛИОТЕКЕ D3DX.....</b>	<b>151</b>
ОБЪЕКТ ID3DXBUFFER .....	151
СТАНДАРТНЫЕ СЕТКИ.....	152
ВИЗУАЛИЗАЦИЯ СЕТКИ .....	154
СКЕЛЕТНЫЕ СЕТКИ .....	155
Загрузка скелетных сеток.....	156
Обновление и визуализация скелетной сетки .....	157
X-СТИЛЬ ТРЕХМЕРНОЙ АНИМАЦИИ.....	158

ТЕХНИКА КЛЮЧЕВЫХ КАДРОВ .....	159
АНИМАЦИЯ В X-ФАЙЛАХ .....	160
<b>ЗАКАНЧИВАЕМ С ГРАФИКОЙ .....</b>	<b>161</b>
 <b>ГЛАВА 3 ВЗАИМОДЕЙСТВИЕ С ПОЛЬЗОВАТЕЛЕМ С</b>	
<b>DIRECTINPUT .....</b>	<b>163</b>
<b>ЗНАКОМСТВО С УСТРОЙСТВАМИ ВВОДА .....</b>	<b>163</b>
ВЗАИМОДЕЙСТВИЕ ЧЕРЕЗ КЛАВИАТУРУ .....	164
Работа с клавиатурой в Windows .....	165
ИГРАЕМ С МЫШЬЮ .....	166
УДОВОЛЬСТВИЕ С ДЖОЙСТИКОМ .....	168
<b>ИСПОЛЬЗОВАНИЕ DIRECTINPUT .....</b>	<b>169</b>
Представляем основы DirectInput .....	169
Инициализация DirectInput .....	171
<b>ИСПОЛЬЗОВАНИЕ УСТРОЙСТВ DIRECTINPUT .....</b>	<b>172</b>
Получение GUID устройства .....	173
Создание COM-объекта устройства .....	177
УСТАНОВКА ФОРМАТА ДАННЫХ .....	177
УСТАНОВКА УРОВНЯ КООПЕРАЦИИ .....	178
УСТАНОВКА СПЕЦИАЛЬНЫХ СВОЙСТВ .....	179
ЗАХВАТ УСТРОЙСТВА .....	181
ОПРОС УСТРОЙСТВА .....	182
ЧТЕНИЕ ДАННЫХ .....	182
<b>DIRECTINPUT И КЛАВИАТУРА .....</b>	<b>183</b>
<b>DIRECTINPUT И МЫШЬ .....</b>	<b>185</b>
<b>DIRECTINPUT И ДЖОЙСТИК .....</b>	<b>186</b>
<b>ЗАКАНЧИВАЕМ С ВВОДОМ .....</b>	<b>191</b>
 <b>ГЛАВА 4 ВОСПРОИЗВЕДЕНИЕ ЗВУКОВ И МУЗЫКИ С</b>	
<b>DIRECTX AUDIO И DIRECTSHOW .....</b>	<b>193</b>
<b>ОСНОВЫ ЗВУКА .....</b>	<b>193</b>
ЦИФРОВАЯ ЗВУКОЗАПИСЬ .....	194
МУЗЫКАЛЬНОЕ БЕЗУМИЕ .....	196
MIDI .....	196
DirectMusic .....	196
MP3 .....	197
<b>ЗНАКОМСТВО С DIRECTX AUDIO .....</b>	<b>197</b>
<b>ИСПОЛЬЗОВАНИЕ DIRECTSOUND .....</b>	<b>198</b>
Инициализация DirectSound .....	200
УСТАНОВКА УРОВНЯ КООПЕРАЦИИ .....	201
УСТАНОВКА ФОРМАТА ВОСПРОИЗВЕДЕНИЯ .....	202
Создание объекта первичного звукового буфера .....	202
Установка формата .....	205
ЗАПУСК ПЕРВИЧНОГО ЗВУКОВОГО БУФЕРА .....	206
ИСПОЛЬЗОВАНИЕ ВТОРИЧНЫХ ЗВУКОВЫХ БУФЕРОВ .....	208
БЛОКИРУЙ И ЗАГРУЖАЙ — ЗАГРУЗКА ЗВУКОВЫХ ДАННЫХ В БУФЕР .....	209
ВОСПРОИЗВЕДЕНИЕ ЗВУКОВОГО БУФЕРА .....	212

ИЗМЕНЕНИЕ ГРОМКОСТИ, ПОЗИЦИОНИРОВАНИЯ И ЧАСТОТЫ .....	212
Регулировка громкости .....	213
Позиционирование .....	213
Изменение частоты .....	214
ПОТЕРЯ ФОКУСА .....	215
ИСПОЛЬЗОВАНИЕ УВЕДОМЛЕНИЙ .....	215
ИСПОЛЬЗОВАНИЕ ПОТОКОВ ДЛЯ СКАНИРОВАНИЯ СОБЫТИЙ .....	219
ЗАГРУЗКА ЗВУКА В БУФЕР .....	221
ПОТОКОВОЕ ВОСПРОИЗВЕДЕНИЕ ЗВУКА .....	224
<b>РАБОТА С DIRECTMUSIC .....</b>	<b>226</b>
НАЧИНАЕМ РАБОТУ С DIRECTMUSIC .....	227
СОЗДАНИЕ ОБЪЕКТА ИСПОЛНИТЕЛЯ .....	228
СОЗДАНИЕ ОБЪЕКТА ЗАГРУЗЧИКА .....	229
РАБОТА С МУЗЫКАЛЬНЫМИ СЕГМЕНТАМИ .....	231
Загрузка музыкальных сегментов .....	231
Загрузка инструментов .....	233
НАСТРОЙКИ ДЛЯ MIDI .....	236
Установка инструментов .....	236
Использование циклов и повторов .....	237
Воспроизведение и остановка сегмента .....	238
Выгрузка данных сегмента .....	239
ИЗМЕНЕНИЕ МУЗЫКИ .....	240
Установка громкости .....	240
Смена темпа .....	242
Захват аудио-канала .....	243
<b>ПРИСОЕДИНЯЕМСЯ К МР3-РЕВОЛЮЦИИ .....</b>	<b>245</b>
ИСПОЛЬЗОВАНИЕ DIRECTSHOW .....	245
ЗАГРУЗКА МЕДИА-ФАЙЛА .....	247
УПРАВЛЕНИЕ ВОСПРОИЗВЕДЕНИЕМ ВАШЕЙ МУЗЫКИ И ЗВУКОВ .....	248
Воспроизведение звука .....	248
Приостановка воспроизведения .....	248
Завершение воспроизведения .....	248
ОБНАРУЖЕНИЕ СОБЫТИЙ .....	249
ОСВОБОЖДЕНИЕ РЕСУРСОВ DIRECTSHOW .....	250
<b>ЗАКАНЧИВАЕМ С МУЗЫКОЙ .....</b>	<b>251</b>
 <b>ГЛАВА 5 РАБОТА С СЕТЬЮ С DIRECTPLAY .....</b>	 <b>253</b>
<b>ЗНАКОМСТВО С СЕТЯМИ .....</b>	<b>253</b>
СЕТЕВЫЕ МОДЕЛИ .....	254
ЛОББИ .....	255
ЗАДЕРЖКА И ЗАПАЗДЫВАНИЕ .....	256
КОММУНИКАЦИОННЫЕ ПРОТОКОЛЫ .....	256
АДРЕСАЦИЯ .....	257
<b>ВВЕДЕНИЕ В DIRECTPLAY .....</b>	<b>258</b>
СЕТЕВЫЕ ОБЪЕКТЫ .....	259
РАБОТА С ИГРОКАМИ .....	260
СЕТЕВЫЕ СООБЩЕНИЯ .....	261
Асинхронная и синхронная работа .....	262

Безопасность.....	263
Гарантированная доставка.....	263
Регулировка потока.....	263
От маленьких байтов к большим словам .....	264
Идентификация приложений по GUID .....	264
<b>ИНИЦИАЛИЗАЦИЯ СЕТЕВОГО ОБЪЕКТА.....</b>	<b>265</b>
<b>ИСПОЛЬЗОВАНИЕ АДРЕСОВ.....</b>	<b>266</b>
Инициализация объекта адреса.....	267
Добавление компонентов .....	267
Установка поставщика услуг .....	269
Выбор порта .....	269
Назначение устройства .....	270
<b>ИСПОЛЬЗОВАНИЕ ОБРАБОТЧИКОВ СООБЩЕНИЙ .....</b>	<b>272</b>
<b>КОНФИГУРИРОВАНИЕ ДАННЫХ СЕССИИ.....</b>	<b>274</b>
Данные сессии сервера.....	274
Данные сессии клиента .....	276
<b>РАБОТА С СЕРВЕРОМ.....</b>	<b>276</b>
Поддержка игроков .....	279
Работа с сообщениями о создании игрока .....	279
Получение имени игрока .....	280
Уничтожение игроков.....	281
Получение данных .....	282
Отправка сообщений сервера .....	284
Завершение сессии на главном узле .....	286
<b>РАБОТА С КЛИЕНТОМ .....</b>	<b>287</b>
Получение и отправка сообщений .....	291
Получение идентификатора игрока .....	291
Завершение сессии клиента .....	292
<b>ЗАКАНЧИВАЕМ С СЕТЯМИ.....</b>	<b>292</b>
 <b>ГЛАВА 6 СОЗДАЕМ ЯДРО ИГРЫ .....</b>	 <b>295</b>
<b>ЗНАКОМСТВО С КОНЦЕПЦИЕЙ ЯДРА .....</b>	<b>295</b>
<b>СИСТЕМНОЕ ЯДРО.....</b>	<b>297</b>
Использование объекта ядра с APPLICATION .....	297
Обработка состояний с STATEMANAGER.....	300
Процессы и с PROCESSMANAGER.....	302
Управление данными с с DATAPACKAGE .....	303
<b>ГРАФИЧЕСКОЕ ЯДРО .....</b>	<b>304</b>
Графическая система и с GRAPHICS .....	305
Изображения и с TEXTURE .....	308
Цвета и с MATERIAL.....	309
Освещение с с LIGHT.....	311
Текст и шрифты с использованием с FONT .....	312
Вершины и с VERTEXBUFFER.....	314
Мировое преобразование с с WORLDPOSITION .....	316
Преобразование вида и с CAMERA .....	318
Загружаемые сетки и использование с MESH .....	320
Рисуем объекты, используя с OBJECT.....	321

ДЕЛАЕМ СЕТКИ ДВИГАЮЩИМИСЯ С CAnimation .....	324
<b>ЯДРО ВВОДА .....</b>	<b>325</b>
ИСПОЛЬЗОВАНИЕ DirectInput С CInput .....	326
УСТРОЙСТВА ВВОДА И CInputDevice .....	326
ИСПОЛЬЗОВАНИЕ ЯДРА ВВОДА .....	329
<b>ЗВУКОВОЕ ЯДРО.....</b>	<b>329</b>
УПРАВЛЕНИЕ DirectX Audio С CSound .....	330
ИСПОЛЬЗОВАНИЕ ВОЛНОВЫХ ДАННЫХ И CSoundData .....	332
ВОСПРОИЗВЕДЕНИЕ ЗВУКА С CSoundChannel.....	334
СЛУШАЕМ МУЗЫКУ С CMusicChannel .....	337
СМЕШИВАНИЕ ИНСТРУМЕНТОВ С CDLS .....	338
ВОСПРОИЗВЕДЕНИЕ MP3 С CMP3 .....	340
<b>СЕТЕВОЕ ЯДРО .....</b>	<b>341</b>
ЗАПРОС АДАПТЕРА С CNetworkAdapter .....	341
СЕРВЕРЫ И CNetworkServer.....	342
РАБОТА КЛИЕНТА И CNetworkClient .....	345
<b>ЗАКАНЧИВАЕМ ИЗУЧАТЬ ЯДРО ИГРЫ .....</b>	<b>347</b>

## **ЧАСТЬ III ПРОГРАММИРОВАНИЕ РОЛЕВЫХ ИГР ..... 349**

### **ГЛАВА 7 ИСПОЛЬЗОВАНИЕ ДВУХМЕРНОЙ ГРАФИКИ ...351**

<b>ЗНАКОМСТВО С БЛОКАМИ И КАРТАМИ .....</b>	<b>351</b>
<b>ИСПОЛЬЗОВАНИЕ БЛОКОВ С DirectX .....</b>	<b>352</b>
ПОСТРОЕНИЕ КЛАССА ДЛЯ РАБОТЫ С БЛОКАМИ.....	356
cTiles::Create .....	357
cTiles::Free .....	358
cTiles::Load .....	358
cTiles::Free .....	360
cTiles::GetWidth, cTiles::GetHeight и cTiles::GetNum.....	360
cTiles::SetTransparent .....	361
cTiles::Draw .....	361
ИСПОЛЬЗОВАНИЕ КЛАССА РАБОТЫ С БЛОКАМИ .....	362
<b>ОСНОВЫ РАБОТЫ С БЛОЧНОЙ ГРАФИКОЙ .....</b>	<b>363</b>
РИСОВАНИЕ ОСНОВНОЙ КАРТЫ .....	363
ИСПОЛЬЗОВАНИЕ НЕСКОЛЬКИХ СЛОЕВ .....	364
ДОБАВЛЕНИЕ ОБЪЕКТОВ .....	365
ПЛАВНАЯ ПРОКРУТКА.....	366
КАРТА И МЫШЬ.....	368
СОЗДАНИЕ КЛАССА КАРТЫ .....	370
<b>ПСЕВДОТРЕХМЕРНЫЕ БЛОКИ.....</b>	<b>375</b>
<b>РАБОТА С БОЛЬШИМИ РАСТРАМИ .....</b>	<b>376</b>
СОЗДАНИЕ БОЛЬШИХ БЛОКОВ .....	377
БОЛЬШОЙ ПРИМЕР .....	377
<b>ЗАКАНЧИВАЕМ С ДВУХМЕРНОЙ ГРАФИКОЙ .....</b>	<b>378</b>

<b>ГЛАВА 8 СОЗДАНИЕ ТРЕХМЕРНОГО ГРАФИЧЕСКОГО ДВИЖКА.....</b>	<b>379</b>
<b>СЕТКИ В КАЧЕСТВЕ УРОВНЕЙ.....</b>	<b>379</b>
ЗАГРУЗКА УРОВНЕЙ .....	380
РИСОВАНИЕ КОМНАТ.....	381
УСОВЕРШЕНСТВОВАНИЕ БАЗОВОЙ ТЕХНИКИ.....	383
<b>ЗНАКОМСТВО С ПИРАМИДОЙ ВИДИМОГО ПРОСТРАНСТВА.....</b>	<b>384</b>
ПЛОСКОСТИ И ОТСЕЧЕНИЕ .....	385
Проверка видимости с плоскостями.....	387
Проверка всей пирамиды.....	387
КЛАСС CFRUSTUM .....	388
cFrustum::Construct.....	389
cFrustum::CheckPoint, CheckCube, CheckRectangle и CheckSphere .....	390
<b>РАЗРАБОТКА ПРОФЕССИОНАЛЬНОГО ТРЕХМЕРНОГО ДВИЖКА .....</b>	<b>393</b>
ЗНАКОМСТВО С ДВИЖКОМ NODETREE.....	394
Создание узлов и деревьев.....	395
Сканирование и рисование дерева .....	397
РАБОТА С ГРУППАМИ МАТЕРИАЛОВ.....	397
СОЗДАНИЕ КЛАССА CNODETREETMESH .....	398
cNodeTreeMesh::Create и cNodeTreeMesh::Free .....	403
cNodeTreeMesh::SortNode .....	406
cNodeTreeMesh::IsPolygonContained и cNodeTreeMesh::CountPolygons .....	408
cNodeTreeMesh::AddNode.....	409
cNodeTreeMesh::Render.....	412
ИСПОЛЬЗОВАНИЕ CNODETREETMESH .....	414
<b>ДОБАВЛЕНИЕ К МИРУ ТРЕХМЕРНЫХ ОБЪЕКТОВ .....</b>	<b>415</b>
ВЫЧИСЛЕНИЕ ОГРАНИЧИВАЮЩЕЙ СФЕРЫ.....	416
ОГРАНИЧИВАЮЩИЕ СФЕРЫ И ПИРАМИДА ВИДИМОГО ПРОСТРАНСТВА.....	417
<b>ОБНАРУЖЕНИЕ СТОЛКНОВЕНИЙ СЕТОК .....</b>	<b>418</b>
СТОЛКНОВЕНИЕ С МИРОМ .....	418
Испускание лучей .....	418
Блокировка пути.....	419
Перемещение вверх и вниз .....	420
Быстрая проверка пересечения.....	422
Обнаружение столкновений в классе cNodeTreeMesh .....	422
КОГДА СТАЛКИВАЮТСЯ СЕТКИ .....	422
ЩЕЛЧКИ МЫШИ И СЕТКИ .....	424
<b>ИСПОЛЬЗОВАНИЕ НЕБЕСНОГО КУБА.....</b>	<b>426</b>
СОЗДАНИЕ КЛАССА НЕБЕСНОГО КУБА .....	427
cSkyBox::Create и cSkyBox::Free .....	428
cSkyBox::LoadTexture .....	429
cSkyBox::Rotate и cSkyBox::RotateRel.....	430
cSkyBox::Render .....	430
ИСПОЛЬЗОВАНИЕ КЛАССА НЕБЕСНОГО КУБА .....	431
<b>ЗАКАНЧИВАЕМ С ТРЕХМЕРНОЙ ГРАФИКОЙ.....</b>	<b>431</b>

<b>ГЛАВА 9 ОБЪЕДИНЕНИЕ ДВУХМЕРНОЙ И ТРЕХМЕРНОЙ ГРАФИКИ .....</b>	<b>433</b>
СМЕШИВАНИЕ ДВУХ ИЗМЕРЕНИЙ.....	433
ИСПОЛЬЗОВАНИЕ ДВУХМЕРНЫХ ОБЪЕКТОВ В ТРЕХМЕРНОМ МИРЕ	434
РИСОВАНИЕ БЛОКОВ В ТРЕХМЕРНОМ МИРЕ.....	434
Загрузка сетки уровня .....	436
Загрузка блоков .....	436
Подготовка к рисованию.....	436
Рисование сетки уровня .....	437
Рисование двухмерных объектов.....	437
ПЕРЕМЕЩЕНИЕ В ТРЕХМЕРНОМ МИРЕ .....	438
ДОБАВЛЕНИЕ ТРЕХМЕРНЫХ ОБЪЕКТОВ К ДВУХМЕРНОМУ МИРУ.....	438
РАБОТА С ДВУХМЕРНЫМ ФОНОМ .....	441
РАБОТА С СЕТКОЙ СЦЕНЫ .....	443
ВИЗУАЛИЗАЦИЯ СЦЕНЫ .....	445
ДОБАВЛЕНИЕ ТРЕХМЕРНЫХ ОБЪЕКТОВ .....	447
СТОЛКНОВЕНИЯ И ПЕРЕСЕЧЕНИЯ.....	447
ЗАКАНЧИВАЕМ СМЕШИВАНИЕ ГРАФИКИ .....	447
<b>ГЛАВА 10 РЕАЛИЗАЦИЯ СКРИПТОВ .....</b>	<b>449</b>
ЧТО ТАКОЕ СКРИПТЫ .....	449
СОЗДАНИЕ СИСТЕМЫ MAD LIB SCRIPT .....	451
ПРОЕКТИРОВАНИЕ СИСТЕМЫ MAD LIB SCRIPT .....	451
ПРОГРАММИРОВАНИЕ СИСТЕМЫ MAD LIB SCRIPT .....	452
Работа с шаблонами действий .....	453
Создание элементов скрипта.....	457
Объединяем все в классе cActionTemplate .....	462
РАБОТА С РЕДАКТОРОМ MLS EDITOR.....	469
ВЫПОЛНЕНИЕ СКРИПТОВ MAD LIB.....	472
ПРИМЕНЕНИЕ СКРИПТОВ В ИГРАХ .....	476
ЗАКАНЧИВАЕМ СО СКРИПТАМИ .....	476
<b>ГЛАВА 11 ОПРЕДЕЛЕНИЕ И ИСПОЛЬЗОВАНИЕ ОБЪЕКТОВ .....</b>	<b>477</b>
ОПРЕДЕЛЕНИЕ ОБЪЕКТОВ ДЛЯ ВАШЕЙ ИГРЫ.....	477
ФОРМА ОБЪЕКТА .....	478
ОПРЕДЕЛЕНИЕ ФУНКЦИИ ОБЪЕКТА .....	479
Оружие.....	481
Броня.....	482
Аксессуары .....	482
Еда.....	482
Коллекции.....	482
Транспорт.....	482
Прочее.....	483
ДОБАВЛЕНИЕ ФУНКЦИЙ К ОБЪЕКТУ .....	483
Категории предметов и значения.....	483



Задание цены предметов.....	484
Флаги предмета.....	484
Ограничения использования.....	485
Присоединение скриптов к предметам.....	486
Сетки и изображения.....	487
Итоговая структура данных предмета.....	487
<b>ГЛАВНЫЙ СПИСОК ПРЕДМЕТОВ.....</b>	<b>488</b>
КОНСТРУИРОВАНИЕ MIL.....	489
ИСПОЛЬЗОВАНИЕ РЕДАКТОРА MIL.....	490
ДОСТУП К ПРЕДМЕТАМ ИЗ MIL.....	493
<b>УПРАВЛЕНИЕ ПРЕДМЕТАМИ С ПОМОЩЬЮ СИСТЕМЫ КОНТРОЛЯ</b>	
<b>ИМУЩЕСТВА.....</b>	<b>493</b>
РАЗРАБОТКА ICS КАРТЫ.....	495
cMapICS::Load, cMapICS::Save и cMapICS::Free.....	498
cMapICS::Add и cMapICS::Remove.....	501
cMapICS::GetNumItems, cMapICS::GetParentItem и cMapICS::GetItem.....	503
Использование класса cMapICS.....	503
РАЗРАБОТКА ICS ПЕРСОНАЖА.....	505
Определение класса cCharICS.....	506
Использование класса cCharICS.....	509
<b>ЗАКАНЧИВАЕМ С ОБЪЕКТАМИ И ИМУЩЕСТВОМ.....</b>	<b>510</b>
<b>ГЛАВА 12 УПРАВЛЕНИЕ ИГРОКАМИ И ПЕРСОНАЖАМИ..</b>	<b>511</b>
<b>ИГРОКИ, ПЕРСОНАЖИ, МОНСТРЫ — О БОЖЕ!.....</b>	<b>511</b>
ОПРЕДЕЛЕНИЕ ПЕРСОНАЖЕЙ В ВАШЕЙ ИГРЕ.....	513
Способности персонажа.....	513
Атрибуты персонажа.....	515
Дополнительные статусы персонажа.....	516
Классы персонажей.....	518
Действия персонажей.....	518
ПЕРСОНАЖИ ИГРОКОВ.....	520
Передвижение игрока.....	520
Управление ресурсами.....	520
Увеличение опыта и силы.....	521
НЕЗАВИСИМЫЕ ПЕРСОНАЖИ.....	522
Персонажи монстров.....	523
Графика персонажей.....	524
ПЕРЕМЕЩЕНИЕ ПЕРСОНАЖЕЙ.....	524
УПРАВЛЕНИЕ ПЕРСОНАЖЕМ ИГРОКА.....	525
Управление направлением.....	526
Управление поворотом.....	528
Управление от первого лица.....	529
УПРАВЛЕНИЕ НЕЗАВИСИМЫМИ ПЕРСОНАЖАМИ.....	530
Ожидание.....	530
Блуждание по области.....	530
Ходьба по маршруту.....	532
Использование маршрутных точек.....	532
Ходьба от точки к точке.....	533

<i>Быстрее Пифагора</i> .....	533
<i>Прохождение маршрута</i> .....	535
Следование за другим персонажем .....	536
Избегание персонажа .....	537
АВТОМАТИЧЕСКОЕ УПРАВЛЕНИЕ ПЕРСОНАЖАМИ .....	538
<b>ОБЩЕНИЕ МЕЖДУ ПЕРСОНАЖАМИ</b> .....	<b>538</b>
ГОВОРЯЩИЕ КУКЛЫ .....	539
Управляемые скриптами говорящие куклы .....	539
Показ диалогов и другого текста .....	540
Класс <i>cWindow</i> .....	541
<i>cWindow::cWindow</i> и <i>cWindow::~cWindow</i> .....	542
<i>cWindow::Create</i> и <i>cWindow::Free</i> .....	543
<i>cWindow::SetText</i> .....	543
<i>cWindow::Move</i> .....	544
<i>cWindow::GetHeight</i> .....	546
<i>cWindow::Render</i> .....	546
Использование <i>cWindow</i> .....	547
<b>СКРИПТЫ И ПЕРСОНАЖИ</b> .....	<b>548</b>
КЛАСС СКРИПТА .....	548
СОЗДАНИЕ ПРОИЗВОДНОГО КЛАССА СКРИПТА .....	549
Производный класс .....	550
ИСПОЛЬЗОВАНИЕ ПРОИЗВОДНОГО КЛАССА .....	553
<b>УПРАВЛЕНИЕ РЕСУРСАМИ</b> .....	<b>553</b>
ИСПОЛЬЗОВАНИЕ ПРЕДМЕТОВ .....	553
ИСПОЛЬЗОВАНИЕ МАГИИ .....	554
ТОРГОВЛЯ И ОБМЕН .....	555
<b>РАБОТА С МАГИЕЙ И ЗАКЛИНАНИЯМИ</b> .....	<b>556</b>
ГРАФИКА ЗАКЛИНАНИЯ .....	556
ФУНКЦИЯ ЗАКЛИНАНИЯ .....	558
Изменение здоровья и маны .....	559
Установка и снятие статусов .....	560
Воскрешение и мгновенная смерть .....	560
Расколдовывание .....	560
Телепортация .....	561
НАЦЕЛИВАНИЕ ЗАКЛИНАНИЙ, СТОИМОСТЬ И ШАНСЫ .....	561
ГЛАВНЫЙ СПИСОК ЗАКЛИНАНИЙ .....	561
СПИСОК ЗАКЛИНАНИЙ .....	564
Определение заклинаний с MSL Editor .....	564
СОЗДАНИЕ КОНТРОЛЛЕРА ЗАКЛИНАНИЙ .....	566
Сетки и <i>sSpellMeshList</i> .....	567
Отслеживание заклинаний с использованием <i>sSpellTracker</i> .....	567
Класс <i>cSpellController</i> .....	569
<i>cSpellController::cSpellController</i> и <i>cSpellController::~cSpellController</i> .....	571
<i>cSpellController::Init</i> и <i>cSpellController::Shutdown</i> .....	571
<i>cSpellController::Free</i> .....	571
<i>cSpellController::GetSpell</i> .....	571
<i>cSpellController::Add</i> .....	572
<i>cSpellController::SetAnimData</i> .....	572

<i>cSpellController::Update</i> .....	572
<i>cSpellController::Render</i> .....	572
Определение жертвы и обработка эффекта заклинания.....	572
ИСПОЛЬЗОВАНИЕ КОНТРОЛЛЕРА ЗАКЛИНАНИЙ .....	572
<b>СРАЖЕНИЯ И ПЕРСОНАЖИ .....</b>	<b>573</b>
ИСПОЛЬЗОВАНИЕ ПРАВИЛ СРАЖЕНИЯ ДЛЯ АТАКИ .....	574
Нанесение удара.....	574
Отклонение атаки.....	575
Обработка ущерба.....	575
ЗАКЛИНАНИЯ В БИТВЕ .....	577
ИНТЕЛЛЕКТ В БИТВАХ .....	579
<b>ПОСТРОЕНИЕ ГЛАВНОГО СПИСКА ПЕРСОНАЖЕЙ.....</b>	<b>579</b>
MCL EDITOR .....	583
ИСПОЛЬЗОВАНИЕ ОПРЕДЕЛЕНИЙ ПЕРСОНАЖЕЙ.....	585
<b>СОЗДАНИЕ КЛАССА КОНТРОЛЛЕРА ПЕРСОНАЖЕЙ.....</b>	<b>585</b>
Сетки и SCHARACTERMESHLIST.....	585
ЗАЦИКЛИВАНИЕ АНИМАЦИИ И SCHARANIMATIONINFO .....	585
ПЕРЕМЕЩЕНИЕ И SROUTEPOINT .....	586
ОТСЛЕЖИВАНИЕ ПЕРСОНАЖЕЙ С SCHARACTER .....	586
КЛАСС SCHARACTERCONTROLLER .....	591
ИСПОЛЬЗОВАНИЕ SCHARACTERCONTROLLER .....	603
<b>ДЕМОНСТРАЦИЯ ПЕРСОНАЖЕЙ В ПРОГРАММЕ CHARS.....</b>	<b>605</b>
<b>ЗАКАНЧИВАЕМ С ПЕРСОНАЖАМИ .....</b>	<b>606</b>
 <b>ГЛАВА 13 РАБОТА С КАРТАМИ И УРОВНЯМИ.....</b>	 <b>609</b>
<b>РАЗМЕЩЕНИЕ ПЕРСОНАЖЕЙ НА КАРТЕ .....</b>	<b>609</b>
СПИСОК ПЕРСОНАЖЕЙ КАРТЫ .....	609
Загрузка списка персонажей карты .....	610
Использование списка персонажей карты в вашей игре .....	611
СКРИПТОВОЕ РАЗМЕЩЕНИЕ.....	612
<b>ИСПОЛЬЗОВАНИЕ ТРИГГЕРОВ КАРТЫ.....</b>	<b>612</b>
СФЕРИЧЕСКИЕ ТРИГГЕРЫ.....	613
КУБИЧЕСКИЙ ТРИГГЕР .....	613
ЦИЛИНДРИЧЕСКИЙ ТРИГГЕР .....	614
ТРЕУГОЛЬНЫЙ ТРИГГЕР .....	614
СРАБАТЫВАНИЕ ТРИГГЕРОВ .....	615
СОЗДАНИЕ КЛАССА ТРИГГЕРА.....	615
<i>cTrigger::cTrigger</i> и <i>cTrigger::~~cTrigger</i> .....	618
<i>cTrigger::Load</i> и <i>cTrigger::Save</i> .....	619
<i>cTrigger::AddTrigger</i> .....	621
<i>cTrigger::AddSphere</i> , <i>cTrigger::AddBox</i> , <i>cTrigger::AddCylinder</i> и <i>cTrigger::AddTriangle</i> .....	622
<i>cTrigger::Remove</i> и <i>cTrigger::Free</i> .....	624
<i>cTrigger::GetTrigger</i> .....	625
<i>cTrigger::GetEnableState</i> и <i>cTrigger::Enable</i> .....	628
<i>cTrigger::GetNumTriggers</i> и <i>cTrigger::GetParentTrigger</i> .....	629
ИСПОЛЬЗОВАНИЕ ТРИГГЕРОВ .....	630
Определение файла триггеров.....	630

Загрузка файла триггеров.....	630
Касание триггера .....	630
<b>БЛОКИРОВАНИЕ ПУТИ БАРЬЕРАМИ .....</b>	<b>631</b>
cBARRIER::SETMESH и cBARRIER::SETANIM.....	634
cBARRIER::RENDER .....	635
ДОБАВЛЕНИЕ БАРЬЕРОВ С cBARRIER .....	636
ИСПОЛЬЗОВАНИЕ КЛАССА БАРЬЕРА .....	637
Создание файла данных барьеров.....	637
Загрузка данных барьеров.....	637
Проверка столкновений с барьерами.....	638
Визуализация барьеров .....	638
<b>ИСПОЛЬЗОВАНИЕ АВТОМАТИЧЕСКИХ КАРТ.....</b>	<b>638</b>
АВТОМАТИЧЕСКИЕ КАРТЫ В ДЕЙСТВИИ.....	639
БОЛЬШИЕ КАРТЫ, МАЛЫЕ КАРТЫ.....	639
ЗАГРУЗКА И ОТОБРАЖЕНИЕ АВТОМАТИЧЕСКОЙ КАРТЫ.....	644
СОЗДАНИЕ КЛАССА АВТОМАТИЧЕСКОЙ КАРТЫ .....	644
cAutomap::cAutomap и cAutomap::~~cAutomap .....	646
cAutomap::Create и cAutoMap::Free .....	646
cAutomap::Load и cAutomap::Save .....	650
cAutomap::GetNumSections и cAutomap::EnableSection .....	651
cAutomap::SetWindow и cAutomap::Render .....	652
ИСПОЛЬЗОВАНИЕ CAUTOMAP .....	654
<b>ЗАКАНЧИВАЕМ С КАРТАМИ И УРОВНЯМИ .....</b>	<b>655</b>
 <b>ГЛАВА 14 СОЗДАНИЕ БОЕВЫХ</b>	
<b>    ПОСЛЕДОВАТЕЛЬНОСТЕЙ .....</b>	<b>657</b>
 ПРОЕКТИРОВАНИЕ ВНЕШНИХ БОЕВЫХ ПОСЛЕДОВАТЕЛЬНОСТЕЙ ..	657
ТЕХНИЧЕСКАЯ СТОРОНА .....	659
РАЗРАБОТКА БОЕВЫХ ПОСЛЕДОВАТЕЛЬНОСТЕЙ .....	661
ГЛОБАЛЬНЫЕ ДАННЫЕ .....	663
cAPP::cAPP.....	664
cAPP::INIT .....	664
cAPP::SHUTDOWN.....	668
cAPP::FRAME.....	668
cAPP::GETCHARACTERAT.....	673
ИСПОЛЬЗОВАНИЕ БОЕВЫХ АРАНЖИРОВOK.....	675
ЗАКАНЧИВАЕМ С БОЕВЫМИ ПОСЛЕДОВАТЕЛЬНОСТЯМИ.....	676
 <b>ГЛАВА 15 СЕТЕВОЙ РЕЖИМ ДЛЯ</b>	
<b>    МНОГОПОЛЬЗОВАТЕЛЬСКОЙ ИГРЫ.....</b>	<b>679</b>
 БЕЗУМНАЯ МНОГОПОЛЬЗОВАТЕЛЬСКАЯ СЕЧА .....	679
ПРОЕКТИРОВАНИЕ МНОГОПОЛЬЗОВАТЕЛЬСКОЙ ИГРЫ.....	680
ДЕМОНСТРАЦИОННАЯ ПРОГРАММА NETWORK GAME.....	682
СОЗДАНИЕ АРХИТЕКТУРЫ МНОГОПОЛЬЗОВАТЕЛЬСКОЙ ИГРЫ .....	684
РАБОТАЕМ ВМЕСТЕ: КЛИЕНТ И СЕРВЕР .....	685
ВЗГЛЯД НА СЕРВЕР .....	688
ВЗГЛЯД НА КЛИЕНТА.....	690

<b>РАБОТА НАД ИГРОВЫМ СЕРВЕРОМ.....</b>	<b>691</b>
ХРАНЕНИЕ ИНФОРМАЦИИ ИГРОКА.....	693
ОБРАБОТКА СООБЩЕНИЙ.....	694
От сообщений DirectPlay к игровым сообщениям.....	696
Очередь сообщений .....	700
Обработка игровых сообщений.....	702
<i>cApp::AddPlayer</i> .....	703
<i>cApp::RemovePlayer</i> .....	705
<i>cApp::PlayerInfo</i> .....	706
<i>cApp::PlayerStateChange</i> .....	706
ОБНОВЛЕНИЕ ИГРОКОВ.....	710
ОБНОВЛЕНИЕ СЕТЕВЫХ КЛИЕНТОВ.....	712
ВЫЧИСЛЕНИЕ ЗАПАЗДЫВАНИЯ .....	712
САМОЕ ТРУДНОЕ ПОЗАДИ! .....	713
<b>РАБОТА НАД ИГРОВЫМ КЛИЕНТОМ.....</b>	<b>713</b>
ОБРАБОТКА ДАННЫХ ИГРОКА .....	714
СЕТЕВОЙ КОМПОНЕНТ.....	716
ОБРАБОТКА СООБЩЕНИЙ.....	719
<i>cApp::CreatePlayer</i> .....	719
<i>cApp::DestroyPlayer</i> .....	720
<i>cApp::ChangeState</i> .....	721
ОБНОВЛЕНИЕ ЛОКАЛЬНОГО ИГРОКА .....	722
ОБНОВЛЕНИЕ ВСЕХ ИГРОКОВ.....	727
КЛИЕНТ ВО ВСЕМ БЛЕСКЕ .....	728
<b>ЗАКАНЧИВАЕМ С МНОГОПОЛЬЗОВАТЕЛЬСКИМИ ИГРАМИ .....</b>	<b>729</b>

## **ЧАСТЬ IV ЗАВЕРШАЮЩИЕ ШТРИХИ ..... 731**

### **ГЛАВА 16 ОБЪЕДИНЯЕМ ВСЕ ВМЕСТЕ В ЗАКОНЧЕННУЮ ИГРУ .....733**

<b>ПРОЕКТИРОВАНИЕ ПРОСТОЙ ИГРЫ.....</b>	<b>733</b>
НАПИСАНИЕ СЦЕНАРИЯ ИГРЫ .....	733
Цель игры The Tower.....	735
РАЗРАБОТКА УРОВНЕЙ.....	735
ОПРЕДЕЛЕНИЕ ПЕРСОНАЖЕЙ .....	739
РАСПРЕДЕЛЕНИЕ ПЕРСОНАЖЕЙ .....	742
СОЗДАНИЕ ПРЕДМЕТОВ И ЗАКЛИНАНИЙ.....	742
РАЗРАБОТКА СКРИПТОВ.....	745
ОПРЕДЕЛЕНИЕ УПРАВЛЕНИЯ .....	748
ПЛАНИРОВАНИЕ ХОДА ИГРЫ .....	749
<b>ПРОГРАММИРОВАНИЕ ПРИМЕРА ИГРЫ.....</b>	<b>751</b>
СТРУКТУРИРОВАНИЕ ПРИЛОЖЕНИЯ.....	753
Конструктор <i>cApp</i> .....	755
Функция приложения <i>Init</i> .....	755
Функция <i>Shutdown</i> .....	758
Обработка кадров в функции <i>Frame</i> .....	758
ИСПОЛЬЗОВАНИЕ ОБРАБОТКИ НА ОСНОВЕ СОСТОЯНИЙ.....	759

---

РАБОТА С КАРТАМИ .....	765
ИСПОЛЬЗОВАНИЕ БАРЬЕРОВ И ТРИГГЕРОВ.....	768
УПРАВЛЕНИЕ ПЕРСОНАЖАМИ.....	768
ПОДДЕРЖКА ТОРГОВЛИ .....	774
ВОСПРОИЗВЕДЕНИЕ ЗВУКОВ И МУЗЫКИ .....	775
ВИЗУАЛИЗАЦИЯ СЦЕНЫ .....	777
ОБРАБОТКА СКРИПТОВ .....	778
СОБИРАЕМ ЧАСТИ.....	781
<b>ЗАВЕРШАЕМ СОЗДАНИЕ ИГРЫ.....</b>	<b>781</b>
 <b>ЧАСТЬ V ПРИЛОЖЕНИЯ.....</b>	 <b>783</b>
 <b>ПРИЛОЖЕНИЕ А. СПИСОК ЛИТЕРАТУРЫ .....</b>	 <b>785</b>
РЕКОМЕНДОВАННОЕ ЧТЕНИЕ.....	785
ПОЛУЧЕНИЕ ПОМОЩИ В WEB.....	788
 <b>ПРИЛОЖЕНИЕ В. СОДЕРЖИМОЕ CD-ROM.....</b>	 <b>791</b>
DIRECTX 9.0 SDK .....	792
ПРОБНАЯ ВЕРСИЯ GOLDWAVE 5 .....	792
ПРОБНАЯ ВЕРСИЯ PAINT SHOP PRO 8.....	793
GAMESPACE LIGHT.....	793
 <b>ПРИЛОЖЕНИЕ С. СЛОВАРЬ ТЕРМИНОВ.....</b>	 <b>795</b>
 <b>АЛФАВИТНЫЙ УКАЗАТЕЛЬ.....</b>	 <b>811</b>



*Моей жене 2Е:  
Любви моей жизни и лучшему другу до конца —  
Ты мой свет.*

## Благодарности

Публикация книги требует большого труда и самоотдачи от всех вовлеченных в этот процесс. В первую очередь я хочу поблагодарить мою семью — жену, 2Е, за ее любящую поддержку; мою мать Пэм (Pam); моего брата Джона (John) за моральную поддержку; моих детей, Майкла (Michael), Джона (John) и Джордана (Jordan) за то, что разрешали мне играть в свои видеоигры откуда я почерпнул несколько замечательных идей; моего второго брата, Джейсона (Jason) за поддержку в трудные времена; и Дженифер (Jennifer) достаточно раздражавшую меня, чтобы показать ей, что писательский труд является (все еще) работой, которой стоит заниматься.

Я благодарю технического редактора книги, Венди Джонс (Wendy Jones) — ваш острый глаз несколько раз уберег меня от ошибок и я очень ценю это. Я особенно благодарен рецензенту издательства, Эми Смит (Emi Smith), за проявленное терпение, которое всегда превращало работу с ней в удовольствие. Спасибо Линде Сейферт (Linda Seifert) за элегантное литературное редактирование и Тариде Анантачай (Tarida Anantachai) за соблюдение сроков завершения проекта.

Я искренне благодарю Уэйна Петерса (Страшилу) (Wayne Peters aka Scarecrow) за великолепные модели для примеров книги и Эндрю Рассела (Andrew Russell) за замечательную музыку изумительно подходящую к примерам игры. Конечно же я был бы небрежным, если бы не поблагодарил Сьюзен Хониуэлл (Susan Honeywell) за ее артистическую переработку моих рисунков.

И, в конце я хочу поблагодарить друзей и членов семьи, которые были со мной на этом пути только душой: моего брата Джеффа (Jeff) — мне жаль, что ты не можешь видеть это; Йена МакЭрдли — как видишь, побуждение написать эту книгу не исчезло; и остальных моих друзей и членов семьи — спасибо за то, что вы есть!





# Об авторе

Карьера Джима Адамса и страсть к программированию начались когда ему было девять лет и он обнаружил, что с помощью компьютера Atari и нескольких строк кода можно реализовать почти все, что был способен вообразить его молодой ум.

Все эти годы благодаря все более и более сложным книгам по программированию и бесчисленным часам, посвященным созданию небольших игр, он двигался от компьютера к компьютеру, пока не достиг мира IBM PC. В то же время он осваивал новые языки программирования — начав с Бейсика, перешел к ассемблеру, затем к Паскалю, к С и, наконец, к C++.

Благодаря своим знаниям и опыту программирования игр и бизнес-приложений Джим сделал карьеру в качестве разработчика игр, писателя и консультанта. Он написал множество статей, является автором нескольких книг по программированию, а также соавтором ряда книг посвященных электронике и программированию.

В настоящее время Джим является владельцем The Collective Mind и занимается программированием и консультациями. Вы можете найти Джима на просторах Интернета, где он часто появляется на различных, связанных с играми, сайтах.



# Введение

Со стертыми пальцами и налитыми кровью глазами вы близки к тому, что тяжкий труд скоро окупится. После сотни часов игры в новейшую компьютерную ролевую игру вы достигли конца. Между вами и победой стоит только огромный злобный дракон. Не стоит волноваться — у вас в запасе есть несколько трюков и скоро этот сосунок узнает кто здесь главный. После финальной апокалиптической битвы ваша миссия закончена — игра побеждена.

Путь был долгим и трудным, но теперь, когда все позади, можно сказать что приключение доставило массу удовольствия. Сценарий был неотразим, графика сногшибательна, звуки и музыка бесподобны. Отдохнув вы можете задаться вопросом, можно ли самому создать такой шедевр. Нечто с броским названием, великолепным сценарием, изящным механизмом сражений из той новой игры с замечательной графикой. «Да», скажете вы, «я могу это сделать!»

Хорошо, мой друг, я здесь чтобы сказать да, вы можете это сделать! Книга «Программирование ролевых игр с DirectX» — это билет, который позволит воплотить ваши идеи в жизнь. На эти страницы я втиснул достаточно информации о программировании и ролевых играх, чтобы помочь вам в создании вашей собственной игры. В книге вы узнаете о том, как создать графическую библиотеку и механизм сражений, управлять игроками, использовать скрипты и предметы, а также о том, как сделать игру многопользовательской.

## О чем эта книга

Эта книга предназначена для программистов, которые хотят познакомиться со специализированной областью разработки ролевых игр (RPG). Я считаю, что ролевые игры — это самый интересный жанр. Информацию о программировании RPG достаточно сложно найти и, чтобы восполнить этот пробел, я написал эту книгу. В ней я разделил ролевую игру на базовые составляющие части. Я беру эти компоненты один за другим, детально рассматриваю их и показываю, как использовать их все вместе в вашем игровом проекте. Чтобы увидеть, о каких элементах я буду говорить, просмотрите раздел «Организация книги».

На страницах книги и на прилагаемом к ней диске CD-ROM, вы найдете примеры программ, созданных на основе информации,

содержащейся в каждой главе. Я создавал эти примеры таким образом, чтобы их можно было легко преобразовать в общецелевые и специфичные для RPG компоненты для вашего проекта. Чтобы ознакомиться с особенностями запуска примеров программ, посмотрите приложение В «Содержимое CD-ROM». Фактически я рекомендую вам посмотреть демонстрационные программы до того, как вы начнете читать книгу. Благодаря этому вы будете знать чего ожидать от книги.

## Для кого предназначена книга

Если вы хотите сделать ваши игры привлекательнее — эта книга для вас. В ней вы найдете полезные трюки и идеи, а также информацию, которая позволит вам начать карьеру программиста ролевых игр.

Я написал эту книгу для новичков и для тех, кто уже имеет небольшой опыт разработки RPG. Суть излагаемых сведений ясна и, независимо от вашего программистского опыта и умений, вы найдете эту книгу ценной.

Итак, если вы интересуетесь программированием ролевых игр или вам нужна помощь в разработке определенного игрового компонента, эта книга для вас.

## Организация книги

Книга разделена на пять частей и каждая часть посвящена отдельному набору тем:

- **Часть I**, «Работа с книгой», поможет вам разогреться, узнать о настройке Windows и установке DirectX, а также о том, как создать хорошую структуру проекта для начала разработки вашей RPG.
- **Часть II**, «Основы DirectX» посвящена обсуждению замечательной библиотеки DirectX и тому, как вы можете использовать ее в своих собственных игровых проектах. Вы узнаете как рисовать с DirectX Graphics, воспроизводить звук с DirectX Audio, работать с сетью с DirectPlay и взаимодействовать с пользователем с DirectInput.
- **Часть III**, «Программирование ролевых игр», содержит весь относящийся к программированию RPG код, который я поместил в эту книгу. Темы включают разработку библиотек для двухмерной и трехмерной графики, управление игровыми персонажами, использование скриптов и предметов, а также создание многопользовательской игры.
- **Часть IV**, «Завершающие штрихи», поможет вам завершить проект. В ней вы узнаете как, используя приведенную в книге информацию, можно создать законченную игру. Учтите

приведенные в ней сведения при начале и окончании собственного проекта.

- **Часть V**, «Приложения» представит вам ряд книг и ссылок на полезные сайты, краткий список терминов, используемых в этой книге, а также обзор содержимого прилагаемого к книге CD-ROM.

## Содержимое CD-ROM

Приложение В, «Содержимое CD-ROM», содержит список программ, находящихся на прилагаемом к книге CD-ROM; однако я не могу устоять перед соблазном уже сейчас хотя бы мельком показать вам то, что вы обнаружите на диске. Первое и самое главное — это комплект разработчика Microsoft DirectX 9.0 SDK и полный исходный код всех демонстрационных программ из этой книги.

DirectX является лидером среди библиотек для разработки игр и именно его я использую в этой книге. Перед чтением книги уделите время для установки DirectX на вашей системе. Все необходимые сведения об установке DirectX и подготовке вашего компилятора к работе с DirectX приведены в главе 1.

Помимо DirectX и исходных кодов, CD-ROM содержит много полезных программ. «Каких программ?» — спросите вы. Для начала, как насчет Caligari's gameSpace Light? Верно; вы можете провести тест-драйв новейшей и самой мощной программы моделирования! Но это еще не все. Полный DirectX 9.0 SDK, пробная версия Paint Shop Pro и еще множество программ — все это на одном маленьком круглом диске!

## Типографские соглашения

В книге используется специальное оформление для того, чтобы выделить важную или интересную информацию:

---

**ВНИМАНИЕ!**

В таких врезках я буду предупреждать вас о возможных проблемах.

---

---

**ПРИМЕЧАНИЕ**

Примечания содержат полезную дополнительную информацию к тому материалу, который вы читаете.

---

---

**СОВЕТ**

Советы предлагают техники и решения, делающие программирование проще.

---

## Что нужно для начала

В самом начале вам потребуется установить Microsoft DirectX 9.0 Software Developer Kit, который находится на прилагаемом к книге CD-ROM (либо вы можете загрузить его с сайта Microsoft <http://msdn.microsoft.com/directx>). Этапы установки DirectX описаны в главе 1.

Также вам потребуется компилятор C++; я рекомендую Microsoft Visual C++. Хотя вы и сможете скомпилировать приведенный в книге код с помощью практически любого компилятора C++, код, который относится к DirectX создавался именно для Visual C++ версии 6.0 или выше.

Помимо этих двух вещей вам необходимы цель и мотивация! Хотя задача создания игры и выглядит устрашающе, эта книга даст вам все знания, необходимые для ее выполнения — и помните, игроки ждут ваш шедевр!

# **Часть I**

## **Работа с книгой**

### **Глава 1      Подготовка к работе с книгой**





# Глава 1

## Подготовка к работе с книгой

Добро пожаловать и примите мои поздравления! Если вы читаете эти слова, я могу смело предположить, что вы готовы научиться тому, как создать ту самую замечательную ролевую игру, о которой вы всегда мечтали. Но перед тем, как погрузиться в сложности программирования RPG, давайте сперва подготовим все, что вам понадобится при чтении этой книги.

В этой главе вы узнаете о следующих вещах:

- Установка DirectX.
- Настройка компилятора для использования DirectX.
- Основные сведения о состояниях и процессах.
- Способы хранения игровых данных.
- Построение каркаса приложения.

### DirectX

Давным давно разработчики должны были полагаться на свое знание DOS, чтобы выдаивать из компьютеров всю вычислительную мощь до капли, дабы их игры работали плавно. Windows тогда была ориентированной на бизнес-приложения платформой, а в качестве игровой системы была нежизнеспособна.

Через некоторое время вышла Windows 95, а затем появился DirectX (продукт, помогающий программистам в разработке игр) и Windows ворвалась на сцену разработки игр. Не имея больше причин соблюдать ограничения, накладываемые DOS (древней операционной системой во многом близкой Unix и Linux), программисты медленно переходили к созданию программ, работающих только под управлением Windows. Итак, мой программирующий друг, чтобы выжить в качестве программиста игр, вы должны знать основы программирования для DirectX.

Для этого и предназначена данная книга — она покажет вам основы (я действительно подразумеваю основы) использования DirectX в ваших собственных игровых проектах. Вы хотите знать, что такое DirectX? Хорошо...

Согласно введению из документации DirectX Software Development Kit (DX SDK) Microsoft DirectX — это набор низкоуровневых интерфейсов программирования приложений (API) для создания игр и других высокопроизводительных мультимедийных приложений. Они включают поддержку двумерной (2D) и трехмерной (3D) графики, звуковых эффектов и музыки, устройств ввода и работы с сетью, например, для многопользовательских игр.

Как утверждается в SDK, DirectX является набором программных интерфейсов, которые помогут вам создавать высокопроизводительные игры и приложения. Теперь позвольте мне сказать чем DirectX не является.

DirectX не является пакетом для создания игр; он просто помогает в разработке приложений предоставляя вам API созданные для прямого взаимодействия с аппаратурой компьютера. Если оборудование поставляется вместе с драйверами для DirectX, вы получите доступ к поддерживаемым оборудованием возможностям аппаратного ускорения. Если функции аппаратного ускорения не доступны, DirectX имитирует их программно.

Это означает, что вы получаете согласованный интерфейс с которым работаете, и вам можно больше не беспокоиться о таких вещах, как возможности установленного оборудования. Если возможность не поддерживается установленным устройством, остается вероятность, что программа будет работать благодаря функциям программной эмуляции в DirectX. Ни суеты, ни путаницы; просто программируйте игру и будьте уверены, что она заработает на большинстве систем.

Новые версии DirectX выходят достаточно часто, и в них появляются новые возможности и улучшаются старые. Во время написания книги последней версией была девятая и именно на ней основан материал. В DirectX 9 входят следующие основные компоненты:

- **DirectX Graphics** — система для трехмерной графики.
- **DirectX Audio** — система для работы с музыкой и звуковыми эффектами.
- **DirectPlay** — простейшая функциональность для работы с сетью (Интернет).
- **DirectInput** — простой доступ к клавиатуре, мыши и джойстику.

DirectX SDK поставляется вместе с различными вспомогательными классами и библиотеками, такими как D3DX, делающими использование DirectX проще, за счет предоставления вам ряда удобных классов для работы. D3DX — замечательная библиотека и я буду использовать ее в этой книге везде где возможно.

Я не буду делать вид, что DirectX является чем то большим, чем он есть на самом деле. Как я уже упоминал, это только метод доступа к низкоуровневым функциям, а не пакет для создания игр. Теперь, когда вы уяснили этот факт, давайте перейдем к полезным занятиям — установим DirectX на вашей системе и настроим ваш компилятор для начала работы с SDK.

## Подготовка DirectX

Перед началом работы с кодом и примерами из этой книги вы должны установить и настроить Microsoft DirectX 9 Software Developers Kit (SDK) на своем компьютере. Здесь есть два этапа — установка библиотек времени выполнения и установка компонентов SDK. Программа установки DirectX выполнит для вас эти два этапа вместе, а я упоминаю об этом, чтобы вы видели чего следует ожидать.

---

**ПРИМЕЧАНИЕ**

Первый этап в программировании игр — это установка библиотек времени выполнения Microsoft DirectX и комплекта разработчика программного обеспечения (SDK); оба компонента упакованы в установочный комплект SDK, находящийся на прилагаемом к книге CD-ROM.

---

Библиотеки времени выполнения являются сердцем и душой компонентов DirectX. Библиотеки содержат код, зависящий от установленного в вашей системе оборудования. Создаваемые производителями оборудования, эти библиотеки либо распространяются как часть стандартного комплекта поставки DirectX, либо их можно получить у производителя (на установочном диске или через Интернет). Разработчики найдут в SDK исходные коды, заголовочные файлы и библиотеки. Установите эти файлы, настройте компилятор и можно двигаться дальше. Для разработчиков библиотеки поставляются в двух версиях: отладочной и дистрибутивной. Конечным пользователям (то есть тем людям, которые будут играть в вашу игру) необходимы только дистрибутивные версии. В чем же различие между дистрибутивными и отладочными версиями — спростите вы. Читайте дальше и узнаете.

## Отладочные и дистрибутивные версии

Когда вы устанавливаете SDK, вас спросят, какие версии библиотек вы хотите использовать — отладочные или дистрибутивные. Это важный вопрос, и вы должны понимать, что вам дает каждая из версий.

С одной стороны отладочная версия позволяет вам видеть что творится за кулисами сцены, но за счет скорости и размера. Дистрибутивная версия позволяет работать на полной скорости, но если в программе вдруг возникнет сбой, вы останетесь в неведении относительно вызвавших его причин. Я рекомендую использовать отладочные версии пока вы будете знакомиться с DirectX, и перейти к дистрибутивным версиям, когда вы приобретете опыт и вам будет нужна максимально возможная скорость.

## Установка DirectX 9

Настало время приступить к делу. Вставьте прилагаемый к книге CD-ROM в привод вашего компьютера. Будет показана страница с лицензионным соглашением к книге «Программирование ролевых игр с DirectX, 2-е издание». Для продолжения щелкните по кнопке **I Agree**. Если интерфейс

не появляется автоматически, вы можете запустить его вручную, выполнив следующие действия:

1. На панели задач Windows щелкните по кнопке **Start (Пуск)**, в открывшемся меню выберите пункт **Run (Выполнить)**, и в появившемся на экране диалоговом окне введите d:\start\_here.html (где d: — это буква, назначенная в вашей системе приводе CD-ROM). Вместо этого вы можете запустить Windows Explorer (не Internet Explorer!) обычно расположенный в папке с программами меню **Start**. В расположенном слева списке папок щелкните по значку вашего CD-ROM и в открывшемся списке файлов дважды щелкните по файлу start\_here.html.
2. Когда появится страница с лицензионным соглашением к книге «Программирование ролевых игр с DirectX, 2-е издание» щелкните для продолжения по кнопке **I Agree**. На экран будет выведен интерфейс работы с CD-ROM (показанный на рис. 1.1). Этот интерфейс предлагает вам различные возможности от просмотра кода до установки программ.



*Рис. 1.1. Использование интерфейса CD-ROM для навигации по CD.  
Просматривайте исходный код или устанавливайте утилиты, приложения  
или игры*

3. Для начала процесса установки щелкните по кнопке **DirectX**, а затем щелкните **Install DirectX 9 SDK**.

## Установка DirectMusic Producer

Хотя DirectMusic Producer и не является частью стандартной поставки DirectX SDK, он находится на сопроводительном диске CD-ROM (посмотрите приложение В, «Содержимое CD-ROM», чтобы узнать где именно находится этот пакет). Программа является утилитой для создания звуковых файлов во внутреннем формате DirectMusic (включая импорт MIDI-файлов), которые могут быть воспроизведены с помощью техники, описываемой в главе 4 «Воспроизведение звуков и музыки с DirectX Audio и DirectShow».

Для установки DirectMusic Producer можно воспользоваться программой установки содержимого CD-ROM, либо можно открыть расположенную на компакт-диске папку \DirectX\DirectMusic Producer и запустить программу Setup.exe. Далее следуйте инструкциям для установки и настройки согласно вашим предпочтениям.

## Включаемые файлы и библиотеки

Когда DirectX 9 и SDK установлены и корректно работают (убедиться в этом можно запустив какую-либо из прилагаемых к этой книге демонстрационных программ), вы готовы к добавлению в проект необходимых библиотек и заголовочных файлов.

В следующем разделе вы узнаете как выполнить настройку вашего компилятора, а сейчас взгляните на таблицу 1.1, где приведен список используемых в этой книге компонентов DirectX, а также включаемых файлов и библиотек которые надо указать в параметрах компиляции при создании нового проекта.

**Таблица 1.1.** Компоненты DirectX, включаемые файлы и библиотеки

<i><b>Компонент</b></i>	<i><b>Включаемые файлы</b></i>	<i><b>Файлы библиотек</b></i>
Direct3D	d3d9.h	d3d9.lib
D3DX	d3dx9.h	d3dx9.lib
DirectInput	dinput.h	dinput8.lib
DirectSound	dsound.h	dsound.lib
DirectMusic	dmusici.h	dsound.lib
DirectPlay	dpaddr.h, dplay8.h	dplay.lib
DirectShow	dshow.h	strmiids.lib

## Подготовка компилятора

Компилятор Microsoft Visual C/C++ является мощным инструментальным средством, абсолютно необходимым для написания работающих под управлением Windows приложений на языках C и C++. Текущая, седьмая,

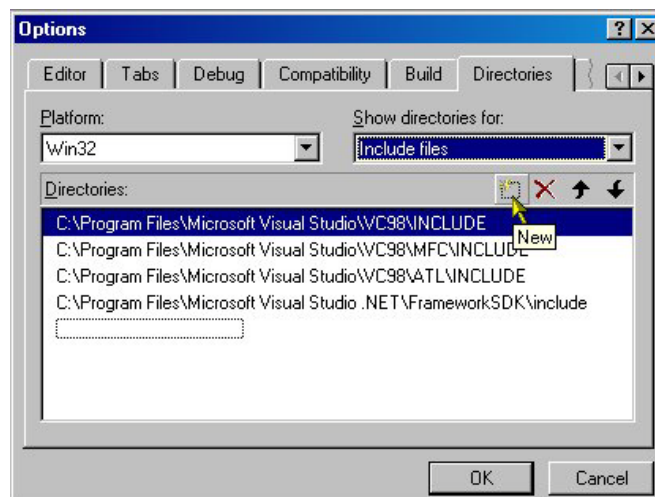
версия (.NET) выбрана множеством разработчиков во всем мире (и выбрана для этой книги). Перед тем как перейти к коду и примерам книги, необходимо задать значения нескольких параметров, чтобы гарантировать правильное выполнение компиляции.

В ходе установки DirectX программа установки DirectX попытается выполнить настройку вашего компилятора, указав пути к SDK, но для завершения настройки необходимо вручную изменить ряд параметров. Процесс настройки и будет рассматриваться в последующих подразделах.

## Указание каталогов для DirectX

Чтобы ваш компилятор смог найти библиотеки и заголовочные файлы DirectX SDK, вы должны добавить соответствующие записи в список каталогов. Те, кто пользуется шестой версией Visual C/C++, должны выполнить следующие действия:

1. Для доступа к списку в главном меню выберите пункт **Tools**, а затем команду **Options**. На экран будет выведено диалоговое окно **Options**.
2. Перейдите на вкладку **Directories** (рис. 1.2).



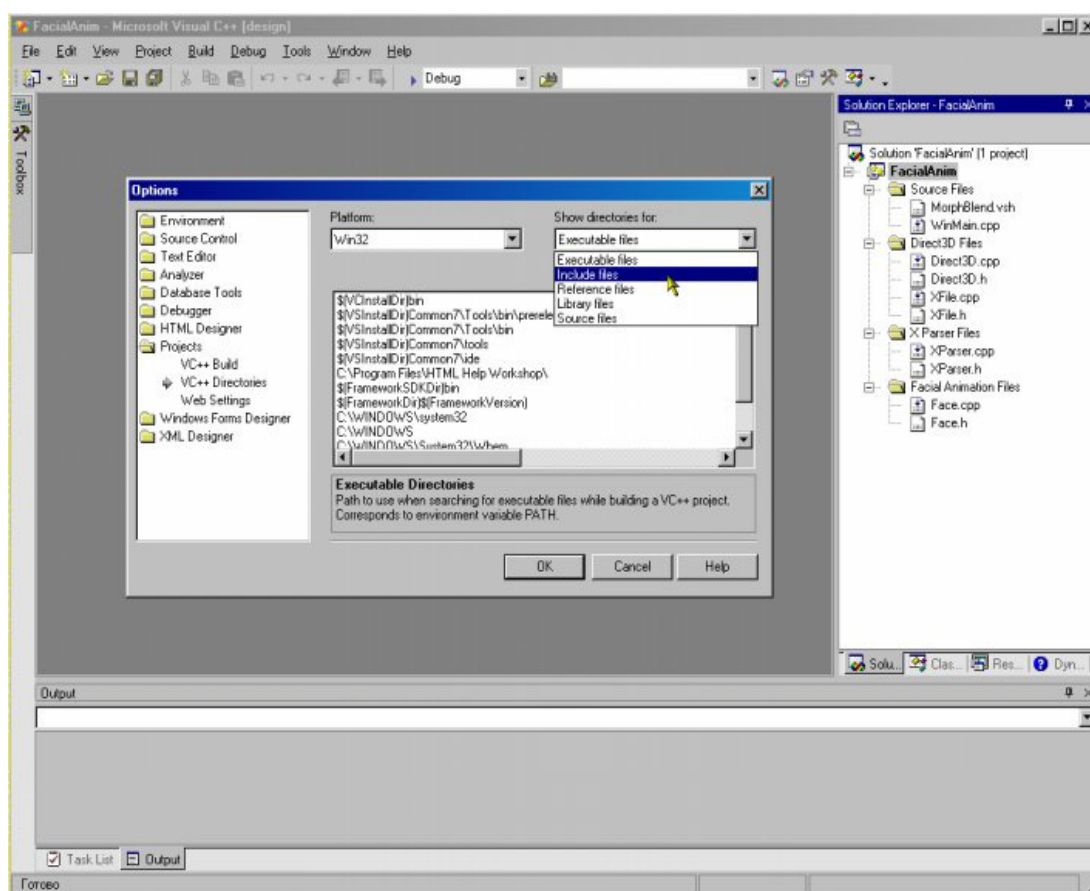
*Рис. 1.2. Вкладка **Directories** диалогового окна **Options** в Visual C/C++ 6, содержащая пути для поиска включаемых заголовочных файлов, библиотек и других файлов с исходным кодом*

3. В выпадающем списке **Show directories for** выберите **Include files**.
4. Если в списке **Directories** не перечислен каталог DirectX, вам надо добавить его. Для этого щелкните по кнопке **New** и введите путь к папке с включаемыми файлами DirectX, выбранный во время установки (проконсультируйтесь с разделом «Установка DirectX 9»). Обычно этот каталог завершается строкой `\include`.
5. Повторите пункт 4 для указания каталога с библиотеками DirectX. Однако, на этот раз в выпадающем списке **Show directories for** выберите **Library files**. Если путь к библиотекам отсутствует в списке, снова щелкните по кнопке **New** и следуйте приведенному в

пункте 4 руководству для указания каталога с библиотеками (он обычно заканчивается на `lib`).

Тем, кто использует Visual Studio .NET, для указания каталогов необходимо выполнить следующие действия:

1. В главном меню выберите пункт **Tools**, а затем команду **Options**. На экран будет выведено диалоговое окно **Options**.
2. В левой части диалогового окна щелкните по папке **Projects**. Папка будет открыта и появятся три подраздела.
3. В подразделах папки **Projects** выберите **VC++ Directories**. В правой части диалогового окна должен появиться список каталогов (как показано на рис. 1.3).



*Рис. 1.3. В Visual Studio .NET выбор узла VC++ Directories приводит к отображению списка каталогов в правой части диалогового окна*

4. В выпадающем списке **Show directories for** выберите **Include files**.
5. Если в списке **Directories** не перечислен каталог DirectX, вам надо добавить его. Для этого щелкните по кнопке **New line** и введите путь к папке с включаемыми файлами DirectX, выбранный во время установки. Обычно этот каталог завершается строкой `include`.
6. Повторите пункт 5 для указания каталога с библиотеками DirectX. Однако, на этот раз в выпадающем списке **Show directories for** выберите **Library files**. Если путь к библиотекам отсутствует в списке, снова щелкните по кнопке **New line** и следуйте



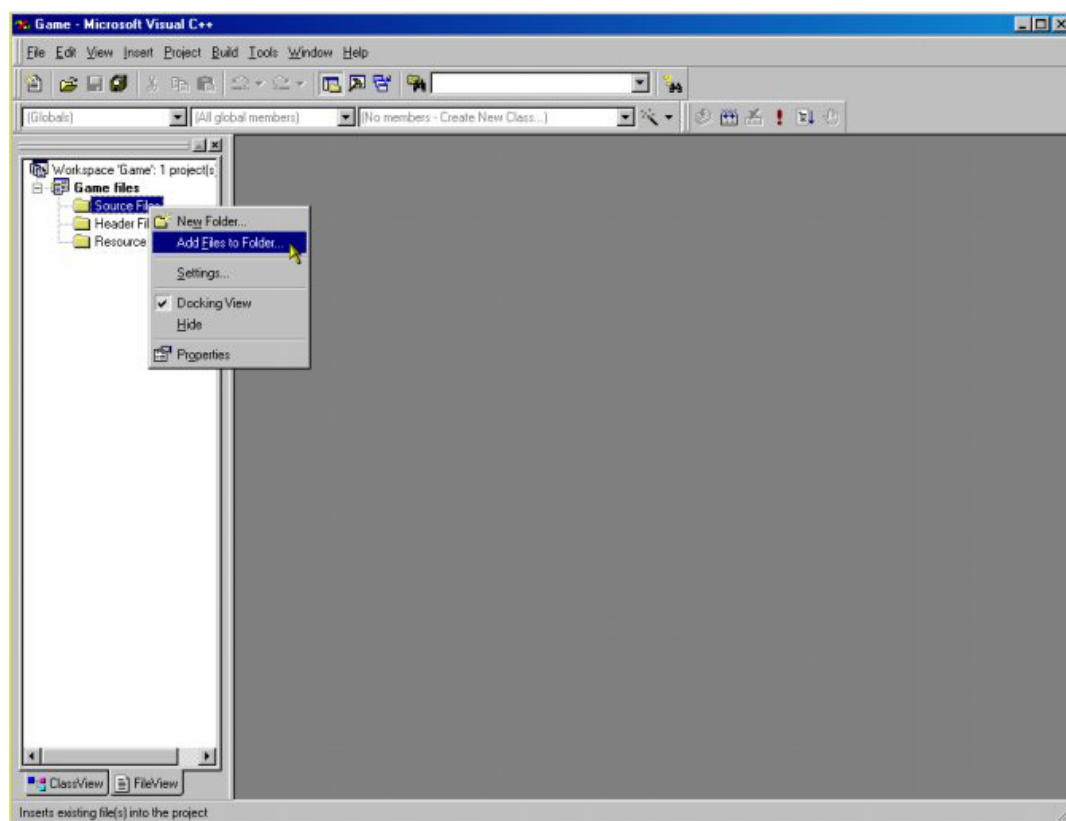
приведенному в пункте 5 руководству для указания каталога с библиотеками (он обычно заканчивается на `\lib`).

## Связывание с библиотеками

Следующий шаг в использовании DirectX (и некоторых других возможностей Windows) — добавление к проекту библиотек. Это можно сделать двумя способами: можно добавить библиотеки в список исходных файлов проекта либо можно добавить файлы библиотек в список **Object/libraries modules**, находящийся в диалоговом окне **Project Settings**.

Чтобы добавить библиотеки в список исходных файлов проекта (в Visual C/C++ 6), выполните следующие действия:

1. Откройте файл проекта и щелкните правой кнопкой мыши по строке **Source Files**, расположенной в окне просмотра рабочего пространства (оно обычно находится в левой части экрана и содержит список включенных в проект файлов). Будет показано небольшое меню (рис. 1.4).

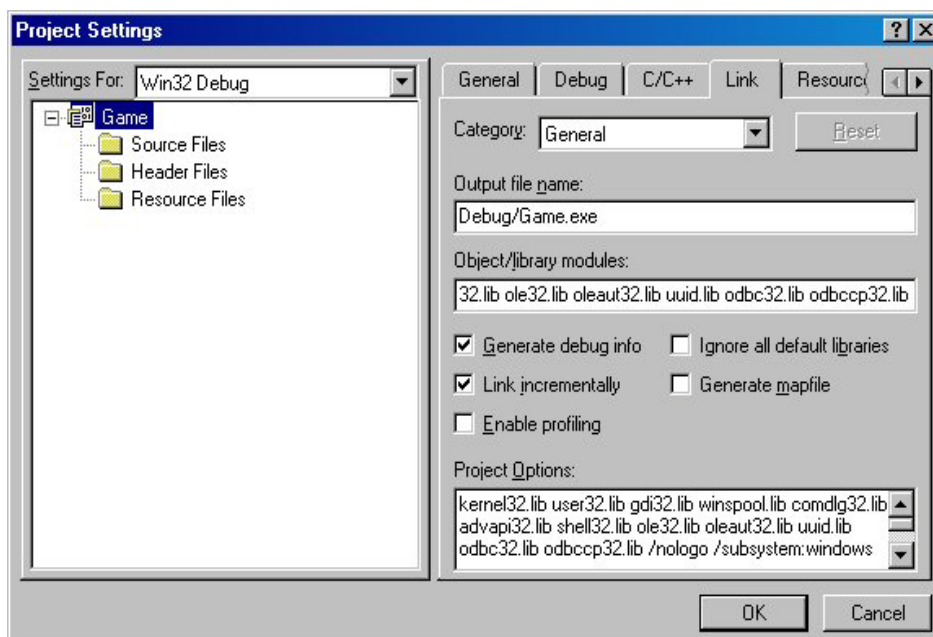


*Рис. 1.4. Добавить файлы библиотек к списку исходных файлов очень легко*

2. Щелкните по пункту **Add Files to Folder**. На экран будет выведено диалоговое окно **Insert Files into Project**.
3. Выберите те библиотеки, которые вы хотите включить в проект и щелкните по кнопке **OK**.

Вы также можете добавить библиотеки в диалоговом окне **Project Settings**, для чего надо выполнить следующие действия:

1. Чтобы получить доступ к параметрам, откройте файл проекта и выберите команду **Settings** из меню **Project**. Будет открыто диалоговое окно **Project Settings**.
2. Перейдите на вкладку **Link**.
3. Если вкладка **Link** отсутствует, убедитесь что в окне со списком **Settings For** выбрано рабочее пространство вашего проекта (рис. 1.5). Рабочее пространство проекта это верхняя строка в окне со списком **Settings For**.

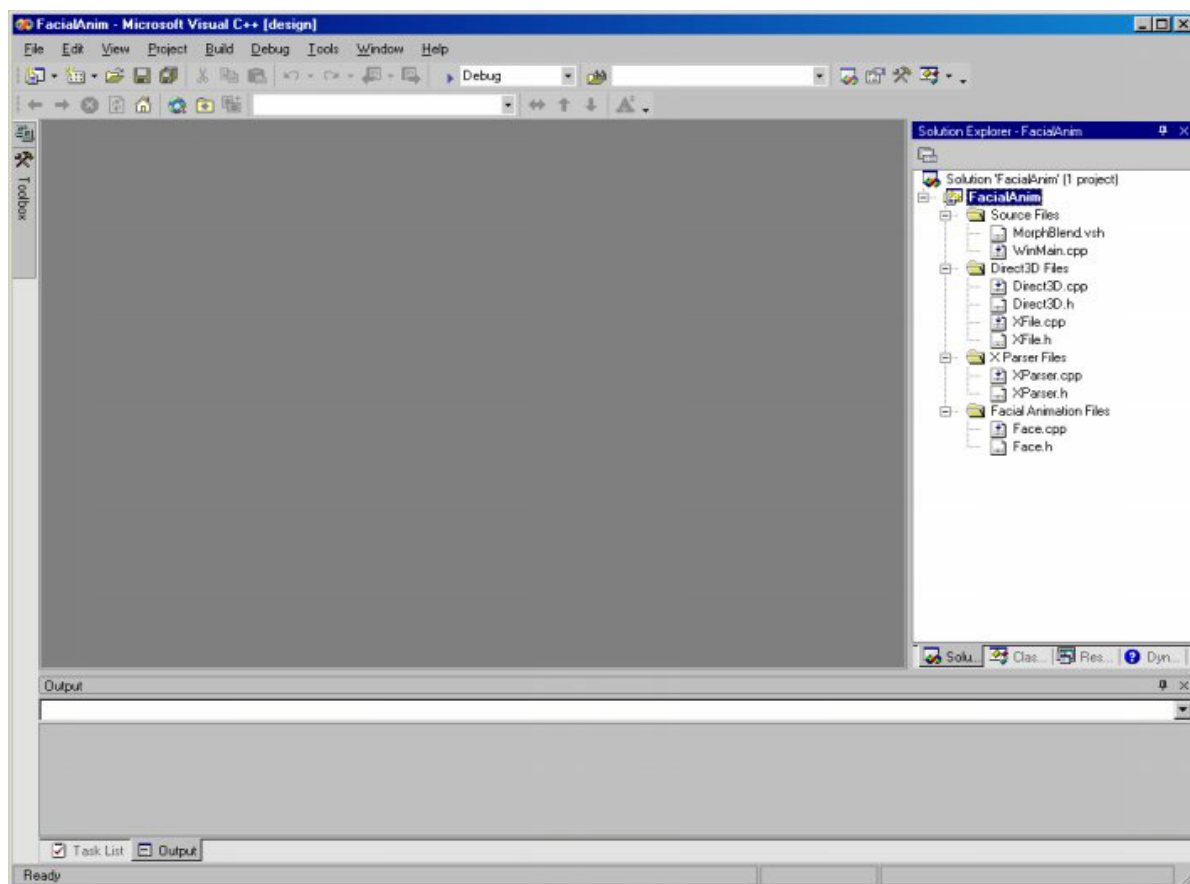


**Рис. 1.5.** Чтобы добавить файлы библиотек в диалоговом окне **Project Settings**, добавляйте имена файлов библиотек в конец строки, находящейся в текстовом поле **Object/library modules**

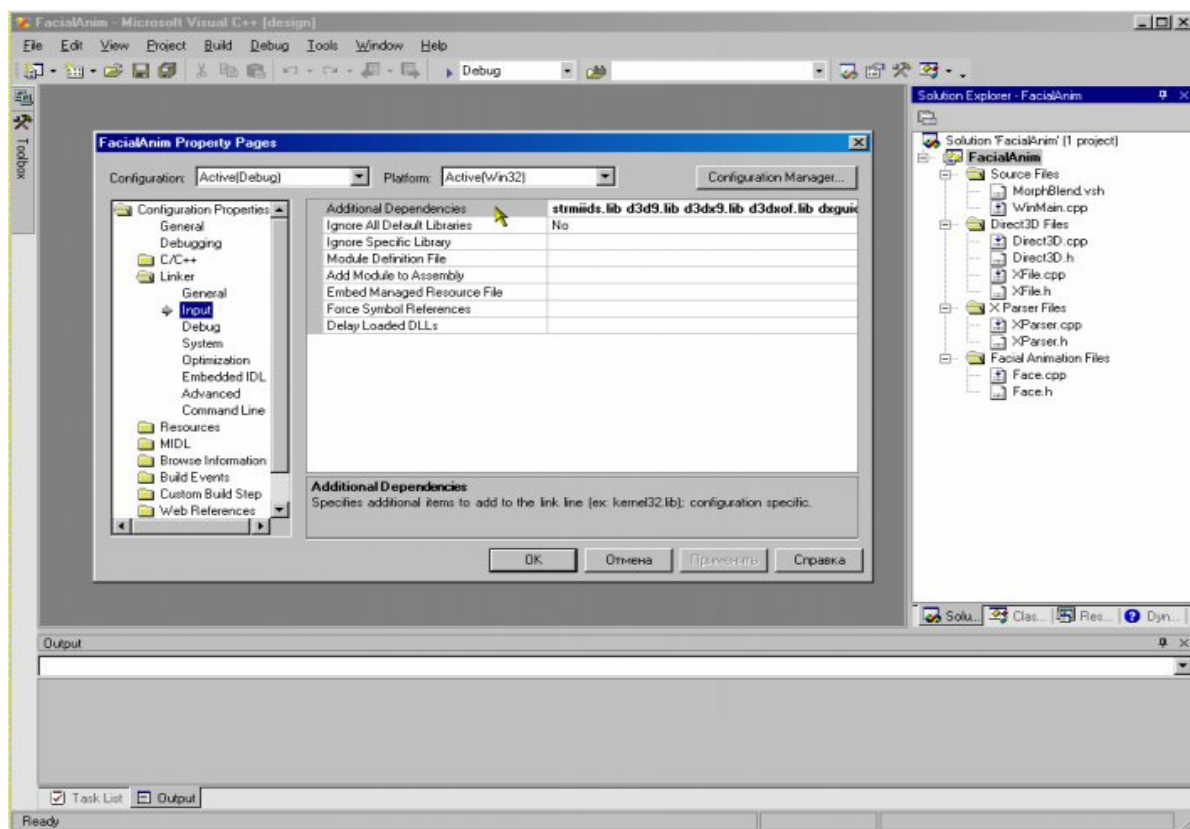
4. В выпадающем списке **Category** выберите **General**. В текстовом поле **Object/library modules** вы увидите список библиотек, которые Visual C/C++ компоует с вашим проектом при компиляции приложения.
5. В конец строки, находящейся в текстовом поле **Object/library modules**, добавьте имена тех библиотек, которые вы хотите компоновать с вашим проектом.

Если вы используете Visual Studio .NET, вам необходимо выделить ваш проект в панели **Solution Explorer** (как показано на рис. 1.6), щелкнуть в главном меню по пункту **Project**, а затем щелкнуть по команде **Properties**. Вам будет показано диалоговое окно **Property Pages** вашего проекта.

В этом окне выберите папку **Linker** и щелкните по строке **Input**. В правой части окна вы увидите параметры компоновки вашего проекта (рис. 1.7).



*Рис. 1.6. В Visual Studio .NET список всех используемых файлов и проектов отображается в окне **Solution Explorer***



*Рис. 1.7. Диалоговое окно **Property Pages** проекта в котором, помимо многих других вещей, вы можете менять параметры компоновки*

В поле **Additional Dependencies** введите имена библиотечных файлов, которые будут указываться в этой книге. Обычно вы должны включить библиотеки D3D9.LIB, D3DX9.LIB, D3DXOF.LIB и WINMM.LIB. Также вам будут нужны и другие библиотеки DirectX, о чем будет отдельно упоминаться в каждой главе.

## Установка поведения по умолчанию для char

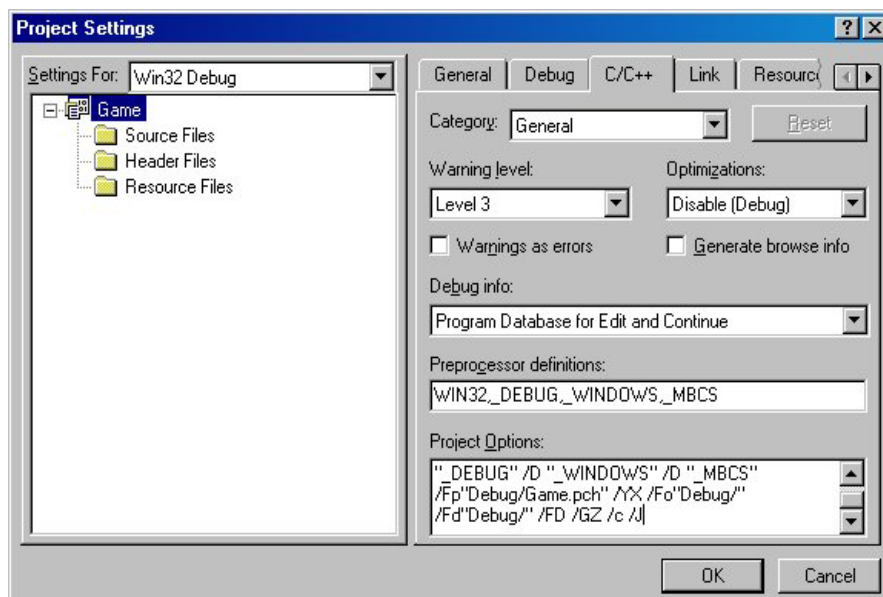
Странно, но в Visual C/C++ отсутствует параметр позволяющий сделать так, чтобы символьные переменные по умолчанию были беззнаковыми. Это означает, что когда вы пишете

```
char Variable;
```

компилятор автоматически расширяет объявление переменной до

```
unsigned char Variable;
```

Долгие годы это было стандартное соглашение об оформлении кода, поскольку многие программисты использовали беззнаковые символьные переменные для хранения чисел из диапазона от 0 до 255. Если писать каждый раз **unsigned char**, будет теряться место, поэтому по умолчанию компилятор преобразовывал объявление **char** в **unsigned char** при компиляции программы. Обычно данный параметр был по умолчанию установлен в конфигурации компилятора.



*Рис. 1.8. Вы можете задать параметры компиляции в текстовом поле Project Options диалогового окна Project Settings*

Однако по каким-то причинам в компиляторе Microsoft Visual C/C++ данный параметр по умолчанию не установлен, а попытка установить его самостоятельно может обескураживать. Чтобы заставить компилятор Visual C/C++ 6 использовать по умолчанию беззнаковые символьные переменные, откройте диалоговое окно **Project Settings** и перейдите на вкладку **C/C++**.

В выпадающем списке **Category** выберите **General**. В текстовом поле **Project Options** добавьте **/J** в конец строки, как показано на рис. 1.8.

В Visual Studio .NET выберите ваш проект в панели **Solution Explorer**, в главном меню выберите пункт **Project** и щелкните по команде **Properties**. В списке папок выберите папку **C/C++** и щелкните по пункту **Command Line**. В поле **Additional Options** добавьте **/J** в конец строки.

## Окончательные и отладочные версии

Для каждого нового проекта Visual C/C++ создает отладочную и окончательную версии, каждую со своим набором параметров. Visual C/C++ делает это по той причине, что в ходе разработки вам может понадобиться указывать особые параметры компиляции для облегчения отладки приложения, не совпадающие с теми параметрами, которые вы используете при сборке окончательной версии приложения. Эти параметры определяют какие ключи компиляции будут установлены по умолчанию и какие библиотеки будет использовать компилятор (например, отладочные версии библиотек времени выполнения).

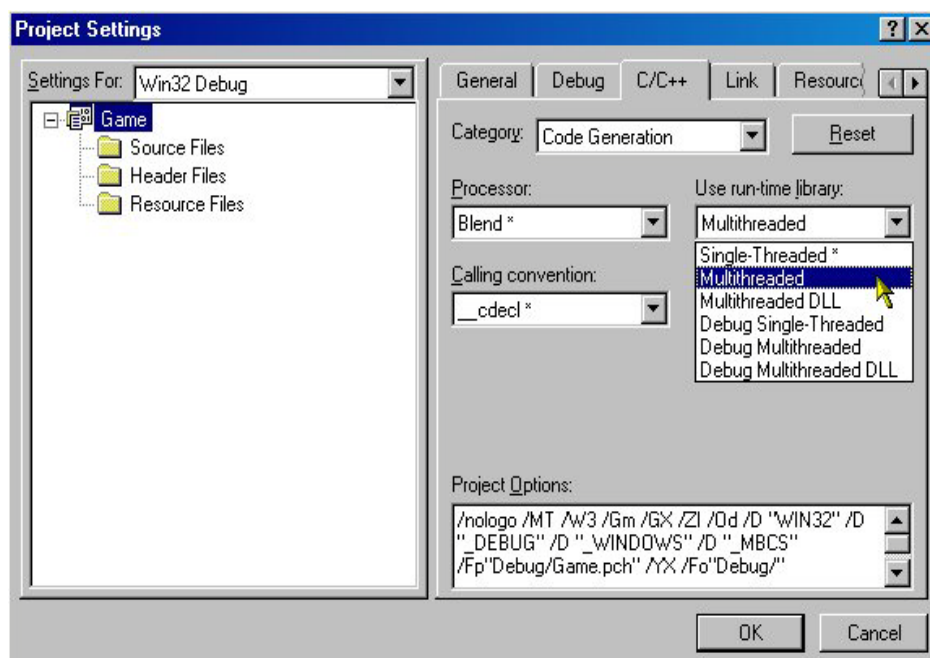
Какую версию (окончательную или отладочную) использовать зависит только от вас. В этой книге я по умолчанию использую отладочные версии и устанавливаемые по умолчанию параметры (за исключением модификации тех параметров, о которых упоминается в этой главе).

## Многопоточные библиотеки

Некоторые компоненты DirectX (например, DirectSound) используют многопоточные библиотеки Windows, поэтому вам надо указать компилятору, чтобы он также использовал их. Для этого в диалоговом окне **Project Settings** (боже, какой занятый диалог!) выполните следующие действия:

1. Откройте диалоговое окно **Project Settings** и перейдите на вкладку **C/C++**.
2. В выпадающем списке **Category** выберите **Code Generation**.
3. В выпадающем списке **Use run-time library** выберите **Multithreaded** (рис. 1.9).

Если вы используете Visual Studio .NET, щелкните по вашему проекту в панели **Solution Explorer**, в главном меню выберите пункт **Project**, а затем команду **Properties**. В окне свойств проекта выберите папку **C/C++** и щелкните по пункту **Code Generation**. В правой части диалогового окна вы увидите выпадающий список **Run-Time Library**. Щелкните по нему и выберите **Multithreaded**.



**Рис. 1.9.** Множество вариантов выбора в выпадающем списке *Use run-time library* может напугать непосвященного. Выберите вариант **Multithreaded** и двигайтесь дальше

## Программирование для Windows

Ну что же, DirectX установлен, ваш компилятор настроен и вы готовы отправиться в путь! Подождите секундочку, мой друг, есть еще несколько вещей на которые следует обратить внимание прежде чем с головой окунуться в мир программирования ролевых игр.

Прежде всего, есть несколько связанных с Windows тем, которые я хочу обсудить. К этим темам относится изучение потоков и многопоточности, критические секции и COM. Я полагаю, что эта информация абсолютно необходима для чтения остальной части этой книги (намек, намек).

### Потоки и многопоточность

Windows 95 познакомила программистов с идеей использования многозадачных систем (даже несмотря на то, что Windows не является истинно многозадачной системой, поскольку использует вытесняющую многозадачность, в которой небольшие фрагменты множества программ выполняются по одному). Идея заключается в том, что у вас есть несколько процессов (приложений) работающих одновременно, каждому из которых выделяется часть времени процессора (называемая *квантом времени* — *time slice*).

Многозадачность позволяет также каждый процесс разделить на несколько отдельных подпроцессов, называемых *потоками* (*threads*). У каждого потока есть своя задача, например, сканирование сетевых данных, обработка пользовательского ввода или воспроизведение звука.



Использование нескольких потоков в одном приложении называется *многопоточностью* (*multithreading*).

Создание дополнительных потоков внутри вашего приложения не составляет труда. Чтобы создать поток вы создаете функцию (используя специальный прототип функции) содержащую код, который вы хотите выполнить. Прототип, используемый для функции потока, выглядит так:

```
DWORD WINAPI ThreadProc(LPVOID lpParameter);
```

Параметр **lpParameter** — это определенный пользователем указатель, который вы предоставляете, когда создаете поток с помощью вызова функции **CreateThread**:

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // NULL
    DWORD dwStackSize, // 0
    LPTHREAD_START_ROUTINE lpStartAddress, // Функция потока
    LPVOID lpParameter, // Предоставляемый
                        // пользователем
                        // указатель
                        // (может быть NULL)
    DWORD dwCreationFlags, // 0
    LPDWORD lpThreadId); // Возвращает
                        // идентификатор потока
```

---

**ВНИМАНИЕ!**

Возвращаемое функцией **CreateThread** значение является дескриптором, который при завершении работы должен быть закрыт, или системные ресурсы не будут освобождены. Освобождение используемых потоком ресурсов выполняется путем вызова функции **CloseHandle**:

```
CloseHandle(hThread); // используйте дескриптор,
                      // возвращенный CreateThread
```

---

Это сложная функция и я не буду углубляться в детали ее работы, а вместо этого приведу пример, который вы можете использовать в качестве образца. В примере показана простая функция потока и вызов для ее инициализации:

```
// Функция пользовательского потока
DWORD WINAPI MyThread(LPVOID lpParameter)
{
    BOOL *Active;
    Active = (BOOL*)lpParameter;
    *Active = TRUE; // флаг активности потока

    // Поместите сюда свой код

    // Завершение потока
    *Active = FALSE; // Сбрасываем флаг активности
    ExitThread(0); // Специальный вызов для закрытия потока
}

void InitThread()
{
    HANDLE hThread;
```

```
DWORD ThreadId;
BOOL Active;

// Создаем поток, передавая определенную пользователем
// переменную, которая используется для хранения
// состояния потока
hThread = CreateThread(NULL, 0,
                      (LPTHREAD_START_ROUTINE)MyThread,
                      (void*)&Active, 0, &ThreadId);

// Ждем завершения выполнения потока,
// проверяя состояние флага
while(Active == TRUE);

// Закрываем дескриптор потока
CloseHandle(hThread);
}
```

Представленный код создает поток, выполнение которого начинается сразу после завершения работы функции **CreateThread**. Создающей поток функции передается указатель на переменную типа **BOOL**, которая отслеживает состояние потока; флаг позволяет определить активность потока и хранит значения **TRUE** (поток активен) или **FALSE** (поток не активен).

Когда выполнение потока завершено, вы сигнализируете о том, что поток больше не является активным (записав значение **FALSE** в упомянутую ранее логическую переменную) и завершаете работу потока вызовом функции **ExitThread**, которой передается единственный параметр — код завершения потока, или, другими словами, число, сообщающее причину по которой выполнение было завершено. В большинстве случаев можно спокойно использовать в вызове **ExitThread** значение 0.

Фактически, поток — это просто функция, выполняемая одновременно с вашим приложением. Более подробно об использовании потоков мы поговорим в главе 4.

## Критические секции

Поскольку Windows — это многозадачная система, отдельные приложения могут мешать друг другу, особенно приложения, использующие несколько потоков. Что если один поток заполняет структуру, содержащую важные данные, а другой поток в это же время хочет изменить или прочитать эти данные?

Есть способ гарантировать, что когда необходимо, только одному потоку будет предоставлен полный контроль, и этот способ — использование *критических секций* (*critical sections*). При активации критическая секция блокирует все другие потоки, пытающиеся получить доступ к разделяемой памяти (памяти приложения, которую используют все потоки), тем самым позволяя потоку в одиночку менять данные приложения, не беспокоясь о возможности вмешательства других потоков.



Чтобы использовать критическую секцию вы должны сперва объявить и инициализировать ее:

```
CRITICAL_SECTION CriticalSection;  
  
InitializeCriticalSection(&CriticalSection);
```

После этого вы можете войти в критическую секцию, обработать данные и покинуть критическую секцию, как показано в приведенном ниже примере:

```
EnterCriticalSection(&CriticalSection);  
  
// Здесь выполняем обработку данных  
  
LeaveCriticalSection(&CriticalSection);
```

Если критическая секция вам больше не нужна (например, когда приложение завершает работу), вы должны освободить ее с помощью вызова:

```
DeleteCriticalSection(&CriticalSection);
```

Хотя о критических секциях можно рассказать и более подробно, реальной необходимости в этом нет. Использовать их достаточно легко и они нужны для многопоточных приложений. Необходимо помнить лишь одно правило — убедитесь, что находящийся в критической секции код выполняется быстро; вы блокируете системные процессы и, если ваша программа выполняется слишком долго, это может привести к краху системы.

## Использование COM

*COM* или *модель компонентных объектов (Component Object Model)* — это применяемая Microsoft техника программирования. Компоненты в парадигме COM — это двоичные объекты, которые могут взаимодействовать между собой используя различные интерфейсы, предоставляемые каждым объектом. Объекты могут заменяться на новые или улучшенные версии не вызывая сбоев в разработанных ранее приложениях. Рассмотрим, к примеру, DirectX — модуль графических объектов DirectX 8 может быть заменен на модуль графических объектов DirectX 9, и вам не надо беспокоиться о том, что ваша, разработанная для DirectX 8 игра больше не будет работать. Чтобы получить больше информации о возможностях COM, посетите посвященный COM сайт Microsoft <http://www.microsoft.com/COM/>.

Используя COM, вы создаете программные компоненты таким образом, чтобы их функциональность была совместима со всеми программами. Возьмем, например, Internet Explorer. Готов поспорить, вы не знаете, что панель инструментов и окно браузера являются COM-объектами. Более того, вы можете использовать эти объекты в своих приложениях!

Хотя это и весома причина для того, чтобы начать использовать COM, более веской причиной является DirectX; DirectX составлен исключительно из COM-объектов.

## Инициализация COM

Для того, чтобы использовать COM-объекты необходимо инициализировать систему COM. Для инициализации COM применяются следующие две функции:

```
// Для однопоточных приложений
HRESULT CoInitialize(
    LPVOID pvReserved); // NULL

// Для многопоточных приложений
HRESULT CoInitializeEx(
    void *pvReserved,    // NULL
    DWORD dwCoInit);     // модель распараллеливания
```

Обе эти функции выполняют поставленную задачу, но если ваше приложение многопоточное, то вы должны использовать вторую функцию, **CoInitializeEx**, поскольку необходимо указать флаг **COINIT\_MULTITHREADED** в параметре **dwCoInit**, для того чтобы система COM функционировала правильно.

Когда вы завершаете использование системы COM, необходимо выключить ее вызовом функции **CoUninitialize**, не имеющей параметров:

```
void CoUninitialize();
```

Каждому вызову **CoInitialize** или **CoInitializeEx** должен соответствовать отдельный вызов **CoUninitialize**. Если вы вызываете **CoInitialize** дважды (это допускается), вам необходимы два вызова **CoUninitialize**. Это иллюстрирует следующий фрагмент кода:

```
// Инициализация системы COM
CoInitialize(NULL);

// Инициализация COM с поддержкой многопоточности
CoInitializeEx(NULL, COINIT_MULTITHREADED);

// Освобождение COM (дважды)
CoUninitialize();
CoUninitialize();
```

## IUnknown

**IUnknown** — это базовый класс для всех COM-интерфейсов. Он содержит только три функции: **AddRef**, **Release** и **QueryInterface**. **AddRef** выполняет, если необходимо, инициализацию, и увеличивает счетчик, показывающий сколько раз был создан экземпляр класса. Вы должны сделать так, чтобы значение счетчика ссылок соответствовало количеству

вызовов функции **Release**, освобождающей ресурсы, используемые экземпляром объекта.

---

<b>ПРИМЕЧАНИЕ</b>	Не путайте классы, интерфейсы и объекты! <i>Объект</i> — это экземпляр <i>класса</i> . <i>Интерфейс</i> — это набор функций, предоставляемых объектом (другими словами, интерфейсы позволяют взаимодействовать с объектом).
-------------------	---

---

Третью функцию, **QueryInterface**, вы используете чтобы получить доступ к предоставляемым объектом интерфейсам, включая их новые версии. Такая ситуация может иметь место, когда было выпущено несколько версий объектов, как в случае DirectX. Вы можете по прежнему использовать старые интерфейсы, а чтобы получить доступ к новым, необходимо запросить их. Если новый интерфейс существует, объект возвратит указатель; в ином случае **QueryInterface** возвращает **NULL**, что свидетельствует об отсутствии интерфейса или об ошибке.

Чтобы добавить функцию, в объекте должен быть класс унаследованный от **IUnknown**, в объявление которого и вставляется код новой функции. Обратите внимание, что согласно установленному Microsoft стандарту COM объекты не могут предоставлять доступ к своим переменным — только к функциям.

Необходимо, чтобы функция возвращала значение типа **HRESULT**, сообщаящее об ошибке или об успешном завершении. Чтобы получить какое-либо значение из COM-объекта вы передаете указатель на переменную (которая должна быть словом или двойным словом — байты и другие типы здесь не поддерживаются) в функцию и использовать этот указатель для возврата значения, находящегося внутри объекта.

В качестве примера создадим простой объект (наследуемый от **IUnknown**), который получает два числа, складывает их и помещает результат в переменную, указанную в третьем параметре:

```
class IMyComObject : public IUnknown
{
    public:
        HRESULT Add(long *Num1, long *Num2, long *Result);
};

HRESULT IMyComObject::Add(long *Num1, long *Num2, long *Result)
{
    // Складываем числа и сохраняем результат
    *Result = *Num1 + *Num2;

    // Возвращаем код успеха
    return S_OK;
}
```

### **Инициализация и освобождение объектов**

Чтобы использовать COM-объект вы должны (помимо написания загружаемой Windows библиотеки) создать его с помощью функции **CoCreateInstance**:

```

STDAPI CoCreateInstance(
    REFCLSID rclsid,          // Идентификатор класса объекта
    LPUNKNOWN pUnkOuter,     // NULL
    DWORD dwClsContext,      // CLSCTX_INPROC
    REFIID riid,             // Ссылка на идентификатор интерфейса
    LPVOID *ppv);           // Указатель для возврата созданного
                           // объекта

```

Для использования функции **CoCreateInstance** необходимо знать идентификаторы класса объекта и интерфейса. Идентификатор класса с префиксом **CLSID\_**, определяет класс объекта, который вы создаете, а ссылка с префиксом **IID\_** — это конкретный интерфейс, который вы ищете. Скажем, у вас есть класс с названием **Math** и идентификатором класса **CLSID\_MATH**. Класс **Math** содержит три объекта: **IAdd** (с идентификатором **IID\_IAdd**), **ISubtract** (**IID\_ISubtract**) и **IAdd2** (**IID\_IAdd2**). Вызов **CoCreateInstance** для получения ссылки на объект **IAdd2** будет выглядеть так:

```

IAdd2 *pAdd2;

if (FAILED(CoCreateInstance(CLSID_MATH, NULL, CLSCTX_INPROC,
                           IID_IAdd2, (void**)&pAdd2))) {
    // Обнаружена ошибка
}

```

Все созданные вами COM-объекты должны быть в конечном счете освобождены. Для этих целей предназначена функция **IUnknown::Release** без параметров:

```

HRESULT IUnknown::Release();

```

После того, как вы завершите работу с интерфейсом **IAdd2**, необходимо освободить его следующим образом:

```

IAdd2->Release();

```

## Запрос интерфейсов

Одна из лучших особенностей COM — обратная совместимость. Если у вас появится новый COM-объект (содержащий новые интерфейсы), то останется полный доступ через объект к старым интерфейсам. Сохранение старых интерфейсов гарантирует, что код будет правильно работать, даже если конечному пользователю установили новые версии COM-объектов. Это также означает, что старые интерфейсы могут запрашивать новые интерфейсы.

Это делается с помощью метода **IUnknown::QueryInterface**:

```

HRESULT IUnknown::QueryInterface(
    REFIID iid,              // Идентификатор нового интерфейса
    void **ppvObject);      // Указатель на новый объект

```

Поскольку исходный объект, вызывающий функцию запроса уже создан, здесь не надо беспокоиться об указании идентификатора класса —

достаточно указать только идентификатор требуемого интерфейса. Вернемся к объекту класса **Math** и предположим, что вы хотите получить интерфейс **IAdd** и затем через него запросить интерфейс **IAdd2**:

```
IAdd *pAdd;
IAdd2 *pAdd2;

// Сперва получаем интерфейс IAdd
if (FAILED(CoCreateInstance(CLSID_MATH, NULL, CLSCTX_INPROC,
                          IID_IAdd, (void**) &pAdd))) {
    // Произошла ошибка
}

// Запрашиваем интерфейс IAdd2
if (SUCCEEDED(pAdd->QueryInterface(IID_IAdd2, (void**) &pAdd2))) {
    // Интерфейс получен, освобождаем первый интерфейс
    IAdd->Release();
}
```

Хотя COM — это обширная тема, приведенной информации достаточно для начала использования DirectX. Вы узнаете больше о DirectX в других главах книги, так что давайте сменим направление и поговорим о других важных для проекта вещах — о потоке выполнения программы.

## Поток выполнения программы

Погрузившись в разработку своего проекта вы можете оказаться погребенными под грудой рутинной работы, такой как модификация кода для того, чтобы он работал с чем-нибудь, что вы добавили, удалили или изменили. Эти работы отнимают драгоценное время, которое лучше потратить на разработку самой игры.

Если вы начнете разработку точно зная свои потребности, то будете в состоянии структурировать ход выполнения операций в программе (называемый *потоком выполнения программы*, *program flow*) и обеспечите легкость внесения изменений. Поскольку вы уже написали проектную документацию (вы ведь сделали это, не так ли?), осталось сделать совсем немного — построить структуру потока обработки.

Обычная программа начинается с инициализации всех систем и данных, а затем переходит к главному циклу. Главный цикл — это то место, где происходит большая часть событий. В зависимости от текущего состояния игры (титульный экран, меню, непосредственно игра и т.д.) вы должны по разному интерпретировать поступающие от пользователя данные и формировать результат их обработки.

Вот этапы, которые проходит стандартное игровое приложение:

1. Инициализация систем (Windows, графика, устройства ввода, звук и т.д.).
2. Подготовка данных (загрузка конфигурационных файлов).
3. Настройка исходного состояния (обычно титульный экран).

4. Запуск главного цикла.
5. Определение состояния и обработка путем получения входных данных, их обработки и формирования выходных данных.
6. Возвращаемся к этапу 5, пока работа приложения не прервана, а потом переходим к этапу 7.
7. Очистка данных (освобождение памяти и т.д.).
8. Освобождение системных ресурсов (Windows, графика, устройства ввода и т.д.).

Шаги с 1 по 3 обычны для всех игр: инициализация различных систем, загрузка необходимых файлов (графических, звуковых и т.д.) и подготовка к запуску самой игры. Большую часть времени ваше приложение будет проводить выполняя внутриигровую обработку (шаг 5), которая может быть разделена на три части: подготовка к обработке кадра, обработка кадра и завершение обработки кадра.

Во время подготовки к обработке кадра выполняется ряд небольших задач, таких как получение текущего времени (для связанных со временем событий, например, синхронизации) и другие частности (например, обновление игровых элементов). Обработка кадра посвящена обновлению объектов (если это не было сделано во время подготовки к обработке кадра) и визуализации графики. Завершение обработки кадра имеет дело с оставшимися функциями, такими как синхронизация по времени или отображение визуализированной графики.

Здесь вас ждет ловушка. В вашей игре может быть несколько состояний обновления кадра: одно для главного меню, другое для самого игрового процесса и т.д. Поддержка нескольких состояний может привести к запутанности кода, но техника, известная как *обработка состояний* (*state processing*) может облегчить бремя. Об обработке состояний мы подробнее поговорим в разделе «Состояния приложения» далее в этой главе.

Очистка данных и выключение систем (шаги 7 и 8) освобождают все системы и ресурсы, выделенные при запуске игры. Необходимо освободить занятую графикой память, уничтожить окно приложения и т.д. Пропускать эти шаги нельзя, так как без них ваша система может остаться в нестабильном состоянии, что приведет к краху!

Каждый этап потока выполнения программы представляется соответствующим блоком кода, поэтому чем лучше структурирован код, тем легче создавать приложение. Для лучшей структуризации кода можно использовать технику программирования известную как *модульное программирование* (*modular programming*).

## Модульное программирование

Модульное программирование является фундаментом многих используемых сегодня техник программирования, включая C++ и COM. В модульном программировании создаются полностью самодостаточные

кодовые модули; им не требуется внешняя поддержка и, в большинстве случаев, они могут работать на различных платформах.

Представьте себе истинно модульную систему программирования в которой написанная вами программа будет работать на всех существующих компьютерах! Вам, вероятно, не придется долго ждать ее появления (возможно, она уже создана). Вы можете думать о модулях программ как о классах C++. Они содержат собственные переменные и функции. Если код написан правильно, классу не требуется внешняя поддержка.

Взяв ваш класс, любое приложение может использовать предоставляемые им возможности, зная только как правильно обращаться к функциям (через использование прототипов функций). Для вызова функции класса достаточно создать экземпляр класса и обратиться к требуемой функции:

```
cClass MyClass;          // Создание экземпляра класса
MyClass.Function1();     // Вызов функции класса
```

Чтобы достигнуть истинной модульности ваш код должен защищать свои данные. Сделать это просто — в C++ достаточно добавить к объявлению переменных классификатор **protected**. Чтобы предоставить доступ к этим переменным класса вы должны написать открытые функции, которые будут использоваться вне вашего кода. Это основы СОМ.

Давайте взглянем на код, иллюстрирующий то, о чем я говорю. Вот класс, содержащий счетчик. Вы можете увеличивать счетчик, устанавливать произвольное значение счетчика и получать текущее значение счетчика — все это с помощью следующего класса:

```
class cCounter
{
    private:
        DWORD m_dwCount;
    public:
        cCounter()
            { m_dwCount = 0; }
        BOOL Increment()
            { m_dwCount++; return TRUE; }
        BOOL Get(DWORD *Var)
            { *Var = m_dwCount; return TRUE; }
        BOOL Set(DWORD Var)
            { m_dwCount = Var; return TRUE; }
};
```

В объявлении класса **cCounter** указано, что переменная **m\_dwCount** является закрытой. Это значит, что даже наследуемые классы не имеют доступа к ней. Остальные функции являются самодокументируемыми. Несколько примечаний требуется только для функции **Get**, которая получает указатель на переменную типа **DWORD**. Функция записывает текущее значение счетчика в указанную переменную и возвращает **TRUE** (как все функции класса **cCounter**).

Это очень простой пример модульного программирования. Более сложным примером является DirectX, который полностью модульный. Если вы хотите использовать только один из компонентов DirectX, скажем DirectSound, то вам необходимо включить только относящиеся к DirectSound объекты. DirectSound не зависит от других компонентов DirectX.

В этой книге я использую техники модульного программирования, создавая ядро независимых друг от друга игровых библиотек. Каждая библиотека объединяет набор уникальных для нее функций — графическая библиотека занимается только обработкой графики, библиотека ввода взаимодействует с устройствами ввода и т.д. Чтобы использовать эти библиотеки достаточно просто включить их в ваш проект и вперед!

## Состояния и процессы

Попытки оптимизировать поток выполнения программы должны быть одной из самых приоритетных задач. Пока ваше приложение маленькое, кодом легко управлять. Однако, как только размер приложения увеличится, работать с ним станет труднее, и даже малейшее изменение потребует переписывания большого объема кода.

Подумайте о следующем: разработка игры идет полным ходом и вы решаете добавить в игру новую возможность — каждый раз, когда пользователь нажимает клавишу **I** будет открываться окно со списком имеющихся у игрока предметов. Экран со списком предметов должен показываться только во время игры, и не должен отображаться, если пользователь находится в меню. Это означает, что вы должны встроить код, обнаруживающий нажатие клавиши **I**, и, когда она нажата, отображать экран со списком предметов вместо обычного игрового экрана.

Если вы решите использовать единственную функцию, которая визуализирует каждый экран в зависимости от того, что пользователь делает в игре (просматривает главное меню, играет или что-либо еще), то быстро обнаружите, что функция визуализации становится очень большой и сложной, поскольку ей необходимо учитывать все состояния, которые могут существовать в игре.

## Состояния приложения

Я упомянул состояния — что это такое? Если коротко, *состояние (state)* — это *образ действий* текущего потока вашего приложения во время выполнения. Главное меню вашей игры — это состояние, точно также как и основной игровой экран тоже состояние. Экран со списком предметов, который вы хотите добавить — это еще одно состояние.

Когда вы начинаете добавлять к своему приложению различные состояния, необходимо предоставить способ определения того, как обрабатывать эти состояния согласно текущему образу действий (который



может меняться по ходу выполнения). Выбор действий, которые ваше приложение должно выполнить для каждого кадра, может привести к появлению такого уродливого кода, как показанный ниже:

```
switch(CurrentState) {
    case STATE_TITLES_SCREEN:
        DoTitleScreen();
        break;

    case STATE_MAINMENU:
        DoMainMenu();
        break;

    case STATE_INGAME:
        DoGameFrame();
        break;
}
```

Подобная конструкция едва работает, особенно если ваша программа перегружена различными состояниями, и практически нежизнеспособна, если вы попытаетесь управлять состояниями для каждого кадра! Вместо этого лучше использовать технику, которую я называю *программированием на основании состояний* (*state-based programming* или, для краткости, *SBP*). Основу этой техники составляет перенаправление потока исполнения на основе стека состояний. Каждое состояние представляется объектом или набором функций. Если вам потребовались функции, вы добавляете их в стек. Когда работа с функциями завершена, они удаляются из стека. Этот процесс показан на рис. 1.10.



**Рис. 1.10.** Стек позволяет помещать и извлекать состояния по мере надобности

Добавлением, удалением и обработкой состояний занимается диспетчер состояний. При добавлении состояния, оно помещается в стек и диспетчер будет передавать управление именно ему. После извлечения из

стека текущее состояние будет удалено и активным станет то состояние, которое располагалось в стеке под ним; именно ему теперь будет передаваться управление.

По упомянутым выше причинам необходимо реализовать диспетчер состояний, который получает указатели на функции (которые представляют состояния). Помещение состояния в стек добавляет на вершину стека указатель на функцию. Ваша задача состоит в вызове диспетчера состояний, который передаст управление верхнему состоянию в стеке. Работа с диспетчером состояний действительно очень проста, так что позвольте мне показать пример, демонстрирующий ее:

```
class cStateManager
{
    // Структура для хранения указателей на функции
    // в связанном списке
    typedef struct sState {
        void (*Function)();
        sState *Next;
    } sState;

protected:
    sState *m_StateParent; // Верхнее состояние в стеке
                           // (голова стека)

public:
    cStateManager() { m_StateParent = NULL; }
    ~cStateManager()
    {
        sState *StatePtr;

        // Удаляем состояния из стека
        while((StatePtr = m_StateParent) != NULL) {
            m_StateParent = StatePtr->Next;
            delete StatePtr;
        }
    }

    // Помещаем функцию в стек
    void Push(void (*Function)())
    {
        // Нельзя помещать нулевой указатель
        if(Function != NULL) {
            // Выделяем ресурсы для нового состояния
            // и помещаем его в стек
            sState *StatePtr = new sState;
            StatePtr->Next = m_StateParent;
            m_StateParent = StatePtr;
            StatePtr->Function = Function;
        }
    }

    // Извлекаем функцию из стека
    BOOL Pop()
    {
        sState *StatePtr = m_StateParent;

        // Удаляем верхнее значение из стека
        // (если оно есть)
        if(StatePtr != NULL) {
```

```
        m_StateParent = StatePtr->Next;
        delete StatePtr;
    }

    // Возвращаем TRUE если в стеке есть еще состояния,
    // и FALSE если стек пуст
    if(m_StateParent == NULL)
        return FALSE;
    return TRUE;
}

// Обрабатываем верхнее сосотояние в стеке
BOOL Process()
{
    // Возвращаем ошибку, если больше нет состояний
    if(m_StateParent == NULL)
        return FALSE;

    // Передаем управление верхнему состоянию
    // (если оно есть)
    m_StateParent->Function();

    return TRUE;
}
};
```

Как видите, класс очень мал, но не позволяйте его размеру одурачить вас. Объект **cStateManager** позволяет вам в любой момент добавлять новые состояния, а в функции визуализации кадра будет достаточно вызвать метод **Process**, и быть уверенным, что управление будет передано требуемой функции.

Вот пример:

```
cStateManager SM;

// Макрос для простого вызова функции MessageBox
#define MB(s) MessageBox(NULL, s, s, MB_OK);

// Прототипы функций состояний - следуйте этим прототипам!
void Func1() { MB("1"); SM.Pop(); }
void Func2() { MB("2"); SM.Pop(); }
void Func3() { MB("3"); SM.Pop(); }

int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hPrev,
                  LPSTR szCmdLine, int nCmdShow)
{
    SM.Push(Func1);
    SM.Push(Func2);
    SM.Push(Func3);

    while(SM.Process() == TRUE);
}
```

В представленной выше небольшой программе есть три состояния, каждое из которых отображает окно сообщений с числом. Каждое состояние извлекает себя из стека и передает управление следующему в стеке состоянию до тех пор, пока все состояния не будут извлечены. Изящно, не так ли?

Представьте себе это предшествование как встроенный в обработку кадра конвейер сообщений. Скажем, вам необходимо отобразить сообщение для пользователя, но сделать это требуется в середине игрового процесса. Нет проблем, поместите функцию отображения сообщения в стек и при формировании следующего кадра игры диспетчер передаст ей управление!

## Процессы

Пойдемте дальше и позвольте мне представить вам еще одну технику, упрощающую вызов функций для каждого кадра. Если вместо непосредственного вызова каждой функции вы используете для управления промежуточными функциями, такими как ввод, работа с сетью и обработка звука, отдельные модули (называемые процессами), можно создать объект, который будет управлять этими модулями за вас.

```
class cProcessManager
{
    // Структура для хранения указателей на функции
    // в виде связанного списка
    typedef struct sProcess {
        void (*Function)();
        sProcess *Next;
    } sProcess;

protected:
    sProcess *m_ProcessParent; // Верхнее состояние в стеке
                                // (голова стека)

public:
    cProcessManager() { m_ProcessParent = NULL; }

    ~cProcessManager()
    {
        sProcess *ProcessPtr;

        // Удаляем все процессы из стека
        while((ProcessPtr = m_ProcessParent) != NULL) {
            m_ProcessParent = ProcessPtr->Next;
            delete ProcessPtr;
        }
    }

    // Добавляем функцию в стек
    void Add(void (*Process)())
    {
        // Нельзя помещать значение NULL
        if(Process != NULL) {
            // Создаем новый процесс
            // и помещаем его в стек
            sProcess *ProcessPtr = new sProcess;
            ProcessPtr->Next = m_ProcessParent;
            m_ProcessParent = ProcessPtr;
            ProcessPtr->Function = Process;
        }
    }

    // Выполняем все функции
    void Process()
    {

```

```

        sProcess *ProcessPtr = m_ProcessParent;
        while(ProcessPtr != NULL) {
            ProcessPtr->Function();
            ProcessPtr = ProcessPtr->Next;
        }
    }
};

```

Снова мы видим простой объект, во многом похожий на объект **cStateManager**, но с одним важным отличием. Объект **cProcessManager** только добавляет процессы; он не может удалять их. Вот пример использования **cProcessManager**:

```

cProcessManager PM;

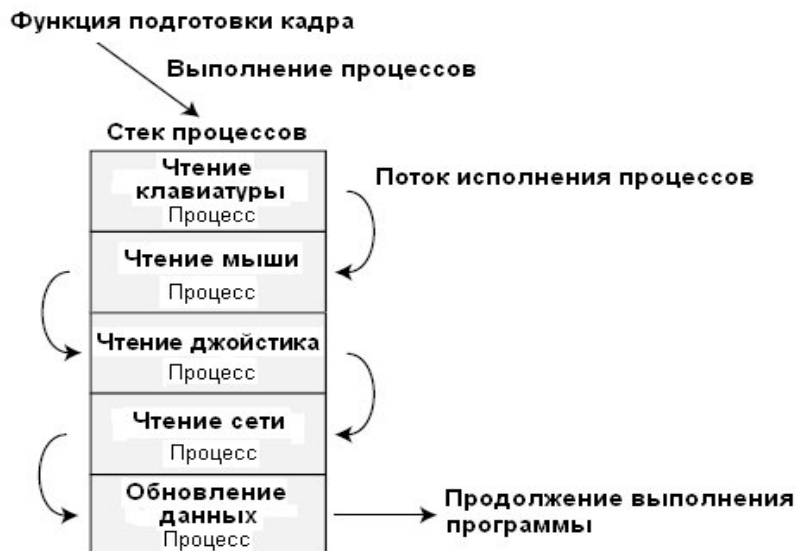
// Макрос, упрощающий вызов функции MessageBox
#define MB(s) MessageBox(NULL, s, s, MB_OK);

// Прототипы функций процессов - следуйте этим прототипам!
void Func1() { MB("1"); }
void Func2() { MB("2"); }
void Func3() { MB("3"); }

int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hPrev,
                  LPSTR szCmdLine, int nCmdShow)
{
    PM.Add(Func1);
    PM.Add(Func2);
    PM.Add(Func3);

    PM.Process();
    PM.Process();
}

```



*Рис. 1.11. Стек процессов состоит из часто вызываемых функций. При вызове **cProcessManager::Process** управление последовательно будет передано каждой добавленной диспетчером функции*

Обратите внимание, что при каждом вызове **Process** вызываются все находящиеся в стеке процессы (как показано на рис. 1.11). Это очень полезно для быстрого вызова часто используемых функций. У вас может

быть несколько объектов диспетчеров процессов для различных ситуаций — например, один для обработки ввода и работы с сетью и другой для обработки ввода и работы со звуком.

## Обработка данных приложения

Все приложения, и особенно игры, в той или иной форме используют данные. Помните вашего персонажа из той компьютерной игры, которой вы посвятили последние три недели? Каждый крошечный фрагмент информации о нем это данные приложения — его имя, очки поражений, уровень опыта, доспехи и оружие, которое он несет. Каждый раз, когда вы выходите из игры, данные вашего персонажа сохраняются и ждут, когда вы снова загрузите их в дальнейшем.

## Использование упаковки данных

Простейший способ обращаться с данными приложения — создать систему упаковки данных, которая поддерживает сохранение и загрузку данных. Создав объект, который содержит буфер для данных, вы можете добавить несколько функций, которые будут сохранять и загружать его для вас. Чтобы лучше понять, о чем я говорю, взгляните на приведенный ниже класс:

```
class cDataPackage
{
    protected:
        // Буфер данных и его размер
        void *m_Buf;
        unsigned long m_Size;

    public:
        cDataPackage() { m_Buf = NULL; m_Size = 0; }

        ~cDataPackage() { Free(); }

        void *Create(unsigned long Size)
        {
            // Освобождаем ранее созданный буфер
            Free();

            // Выделяем память и возвращаем указатель
            return (m_Buf = (void*)new char[(m_Size = Size)]);
        }

        // Освобождаем выделенную память
        void Free() { delete m_Buf; m_Buf = NULL; m_Size = 0; }

        // Сохраняем буфер на диске
        BOOL Save(char *Filename)
        {
            FILE *fp;

            // Проверяем, что есть что-нибудь для записи
            if(m_Buf != NULL && m_Size) {
                //Открываем файл, записываем размер и данные
```

```
        if((fp=fopen(Filename, "wb")) != NULL) {
            fwrite(&m_Size, 1, 4, fp);
            fwrite(m_Buf, 1, m_Size, fp);
            fclose(fp);
            return TRUE;
        }
    }
    return FALSE;
}

// Загружаем данные в буфер из файла
void *Load(char *Filename, unsigned long *Size)
{
    FILE *fp;

    // Освобождаем предыдущий буфер
    Free();

    if((fp=fopen(Filename, "rb")) != NULL) {
        // Читаем размер и данные
        fread(&m_Size, 1, 4, fp);
        if((m_Buf = (void*)new char[m_Size]) != NULL)
            fread(m_Buf, 1, m_Size, fp);
        fclose(fp);

        // Сохраняем размер, чтобы вернуть
        if(Size != NULL)
            *Size = m_Size;

        // Возвращаем указатель
        return m_Buf;
    }
    return NULL;
}
};
```

Класс **cDataPackage** содержит всего четыре функции, которые можно использовать (в действительности их шесть — включая конструктор и деструктор). Первая функция, которую вы будете вызывать, **Create**, выделяет блок памяти указанного вами размера. Функция **Free** освобождает этот блок памяти. Что касается **Save** и **Load**, то они делают следующее — сохраняют блок данных на жестком диске в файле с указанным именем и загружают блок из заданного файла. Обратите внимание, что функции **Create** и **Load** возвращают указатели. Это указатели на буфер данных, для которых вы можете выполнить операцию приведения типа, чтобы преобразовать их в указатели на ваши собственные структуры данных.

## Тестирование системы упаковки данных

Представьте, что вы создали хранящий список имен пакет данных, и хотите использовать для работы с этими именами собственную структуру данных. Создав пакет данных и выполнив приведение типа возвращенного указателя к вашей структуре, вы можете сразу приступить к работе с именами, как показано ниже:

```

// Структура для хранения имени
typedef struct {
    char Name[32];
} sName;

int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hPrev,
                  LPSTR szCmdLine, int nCmdShow)
{
    cDataPackage DP;
    DWORD Size;

    // Создаем пакет данных (размером 64 байта),
    // получаем указатель на него и выполняем
    // приведение типа к sName
    sName *Names = (sName*)DP.Create(64);

    // Поскольку мы выделили 64 байта, а каждое имя
    // использует 32 байта, можно хранить два имени
    strcpy(Names[0].Name, "Jim");
    strcpy(Names[1].Name, "Adams");

    // Сохраняем имена на диске и освобождаем буфер данных
    DP.Save("names.dat");
    DP.Free();

    // Загружаем имена с диска. После возврата из функции
    // загрузки значение Size будет равно 64
    Names = (sName*)DP.Load("names.dat", &Size);

    // Отображаем имена
    MessageBox(NULL, Names[0].Name, "1st Name", MB_OK);
    MessageBox(NULL, Names[1].Name, "2nd Name", MB_OK);

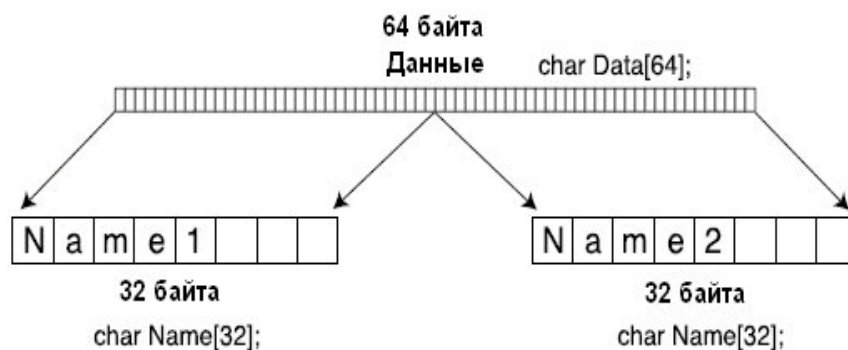
    // Освобождаем пакет данных
    DP.Free();

    return 0;
}

```

Взглянув более пристально на использование буфера данных, вы увидите, что в выделенных 64 байтах для хранения имен используются два блока по 32 байта каждый, как показано на рис. 1.12.

Варианты использования упаковки данных огромны. Создав несколько небольших объектов пакетов данных вы можете выполнять приведение типов для любых указателей, сохраняя все данные вашего приложения в однотипных объектах-контейнерах, поддерживающих сохранение и загрузку.



**Рис. 1.12.** Буфер данных достаточно большой, чтобы хранить все имена. В данном случае хранятся два имени по 32 байта каждое, что дает общий размер буфера равный 64 байтам



## Построение каркаса приложения

Уверен, вы согласитесь, что необходимость постоянно перепечатывать один и тот же код — код для создания окна, рисования графики, воспроизведения звука и т.д., — каждый раз, когда вы начинаете новый проект очень утомляет. Почему бы не составить библиотеку из тех функций, которые присутствуют в каждом приложении, что избавит вас от рутины и оставит больше времени на разработку самого приложения?

Именно эта идея лежит в основе *каркаса приложения* (*application framework*). На базовом уровне каркас должен содержать код для инициализации окна приложения и различных подсистем (графики, ввода, сети и звука), обработки результатов инициализации, выполняемых в каждом кадре процедур и функций для завершения работы приложения. Использование технологии модульного программирования также помогает, поскольку основные компоненты, такие как упомянутые подсистемы, содержатся в отдельных объектах.

На данном этапе цель заключается в разработке простого проекта, который может использоваться в качестве фундамента для остальных ваших приложений. Создайте новый проект и назовите его *framework* (или как-нибудь иначе, но чтобы название описывало его назначение). В этом проекте создайте файл с именем *WinMain.cpp*, который будет представлять точку входа вашего приложения.

Исходный код *WinMain.cpp* должен быть минимальным и содержать только код, необходимый для инициализации окна. Взгляните на исходный код, находящийся в файле *WinMain.cpp*, который я обычно использую в каркасе для своих проектов:

```
// Включаемые файлы
#include <windows.h>
#include <stdio.h>
#include <stdarg.h>

// Экземпляр главного приложения
HINSTANCE g_hInst; // Глобальный дескриптор экземпляра
HWND      g_hWnd;  // Глобальный дескриптор окна

// Размеры окна приложения, его тип, класс и заголовок
#define WNDWIDTH  400
#define WNDHEIGHT 400
#define WNDTYPE   WS_OVERLAPPEDWINDOW

const char g_szClass[]  = "FrameClass";
const char g_szCaption[] = "FrameCaption";

// Главные прототипы функции приложения

// Точка входа
int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hPrev,
                  LPSTR szCmdLine, int nCmdShow);

// Функция для отображения сообщений об ошибках
void AppError(BOOL Fatal, char *Text, ...);
```

```
// Процедура обработки сообщений
long FAR PASCAL WindowProc(HWND hWnd, UINT uMsg,
                           WPARAM wParam, LPARAM lParam);

// Функции для регистрации и отмены регистрации классов
BOOL RegisterWindowClasses(HINSTANCE hInst);
BOOL UnregisterWindowClasses(HINSTANCE hInst);

// Функция для создания окна приложения
HWND CreateMainWindow(HINSTANCE hInst);

// Функции для инициализации, завершения работы и обработки кадров
BOOL DoInit();
BOOL DoShutdown();
BOOL DoPreFrame();
BOOL DoFrame();
BOOL DoPostFrame();

int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hPrev,
                  LPSTR szCmdLine, int nCmdShow)
{
    MSG Msg;

    // Сохраняем экземпляр приложения
    g_hInst = hInst;

    // Регистрация класса окна
    // Возвращаем управление, если не удалась
    if(RegisterWindowClasses(hInst) == FALSE)
        return FALSE;

    // Создание окна
    // Возвращаем управление, если не удалось
    if((g_hWnd = CreateMainWindow(hInst)) == NULL)
        return FALSE;

    // Инициализация приложения
    // Возвращаем управление в случае ошибки
    if(DoInit() == TRUE) {
        // Вход в цикл обработки сообщений
        ZeroMemory(&Msg, sizeof(MSG));
        while(Msg.message != WM_QUIT) {
            // Обработка сообщений Windows (если они есть)
            if(PeekMessage(&Msg, NULL, 0, 0, PM_REMOVE)) {
                TranslateMessage(&Msg);
                DispatchMessage(&Msg);
            } else {
                // Предкадровая обработка,
                // прекращаем обработку кадра,
                // если вернули FALSE
                if(DoPreFrame() == FALSE)
                    break;

                // Обработка кадра,
                // прекращаем обработку,
                // если вернули FALSE
                if(DoFrame() == FALSE)
                    break;

                // Послекадровая обработка,
                // прекращаем обработку кадра,
```

```
        // если вернули FALSE
        if (DoPostFrame() == FALSE)
            break;
    }
}

// Функция завершения работы
DoShutdown();

// Отменяем регистрацию класса окна
UnregisterWindowClasses(hInst);

return TRUE;
}

BOOL RegisterWindowClasses(HINSTANCE hInst)
{
    WNDCLASSEX wcex;
```

---

**ПРИМЕЧАНИЕ**

Вы используете функцию **RegisterWindowClasses** для регистрации класса окна (путем вызова функции **RegisterClassEx**). При вызове функции **RegisterWindowClasses** указывайте в ее единственном параметре значение **HINSTANCE** приложения (то, которое было передано вам в функцию **WinMain**).

---

```
    // Создание и регистрация класса окна
    wcex.cbSize      = sizeof(wcex);
    wcex.style       = CS_CLASSDC;
    wcex.lpfnWndProc  = WindowProc;
    wcex.cbClsExtra  = 0;
    wcex.cbWndExtra  = 0;
    wcex.hInstance   = hInst;
    wcex.hIcon       = LoadIcon(NULL, IDI_APPLICATION);
    wcex.hCursor     = LoadCursor(NULL, IDC_ARROW);
    wcex.hbrBackground = NULL;
    wcex.lpszMenuName = NULL;
    wcex.lpszClassName = g_szClass;
    wcex.hIconSm     = LoadIcon(NULL, IDI_APPLICATION);

    if (!RegisterClassEx(&wcex))
        return FALSE;

    return TRUE;
}

BOOL UnregisterWindowClasses(HINSTANCE hInst)
{
    // Отмена регистрации класса окна
    UnregisterClass(g_szClass, hInst);

    return TRUE;
}

HWND CreateMainWindow(HINSTANCE hInst)
{
    HWND hWnd;
```

**ПРИМЕЧАНИЕ**

Вы используете функцию `CreateMainWindow` для создания и отображения окна заданного размера и типа (указанных макроопределениями `WNDWIDTH`, `WNDHEIGHT` и `WNDTYPE` соответственно). Просто укажите в единственном параметре `CreateMainWindow` значение `HINSTANCE` вашего приложения.

```
// Создание главного окна
hWnd = CreateWindow(g_szClass, g_szCaption,
                   WNDTYPE, 0, 0, WNDWIDTH, WNDHEIGHT,
                   NULL, NULL, hInst, NULL);

if(!hWnd)
    return NULL;

// Отображение и обновление окна
ShowWindow(hWnd, SW_NORMAL);
UpdateWindow(hWnd);

// Возвращаем дескриптор окна
return hWnd;
}

void AppError(BOOL Fatal, char *Text, ...)
{
    char CaptionText[12];
    char ErrorText[2048];
    va_list valist;

    // Создаем заголовок окна сообщения
    // на основе флага Fatal
    if(Fatal == FALSE)
        strcpy(CaptionText, "Error");
    else
        strcpy(CaptionText, "Fatal Error");

    // Создаем текст сообщения
    va_start(valist, Text);
    vsprintf(ErrorText, Text, valist);
    va_end(valist);

    // Отображаем окно сообщений
    MessageBox(NULL, ErrorText, CaptionText,
               MB_OK | MB_ICONEXCLAMATION);

    // Отправляем сообщение о завершении работы приложения,
    // если ошибка является фатальной
    if(Fatal == TRUE)
        PostQuitMessage(0);
}

// Процедура обработки сообщений - обрабатывает только
// сообщение о завершении работы приложения
long FAR PASCAL WindowProc(HWND hWnd, UINT uMsg,
                           WPARAM wParam, LPARAM lParam)
{
    switch(uMsg) {
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
    }
}
```

```
        return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }

    BOOL DoInit()
    {
        // Здесь выполняются функции инициализации приложения, такие как
        // инициализация графической системы, сети, звука и т.д. Возвращает
        // TRUE в случае успешной инициализации и FALSE при ошибке.

        return TRUE;
    }

    BOOL DoShutdown()
    {
        // Здесь выполняются действия, завершающие работу приложения,
        // такие как отключение графики, звука и т.д. Возвращает TRUE
        // в случае успешного завершения работы и FALSE при ошибке.

        return TRUE;
    }

    BOOL DoPreFrame()
    {
        // Выполняет подготовку к формированию кадра, например,
        // установку таймера. Возвращает TRUE в случае успешной
        // подготовки и FALSE при ошибке.

        return TRUE;
    }

    BOOL DoFrame()
    {
        // Выполняет операции формирования кадра, такие как визуализация.
        // Возвращает TRUE при успехе и FALSE при ошибке.

        return TRUE;
    }

    BOOL DoPostFrame()
    {
        // Выполняет посткадровую обработку, например,
        // синхронизацию со временем и т.д.
        // Возвращает TRUE при успехе и FALSE при ошибке.

        return TRUE;
    }
}
```

Представленный выше код каркаса инициализирует окно и запускает цикл обработки сообщений, ожидающий, пока не придет сигнал о завершении программы. Все представленные функции обрабатывают различные моменты инициализации или завершения работы приложения. Обратите внимание, что создаваемое окно приложения не является фоновым — это необходимо для работы с DirectX Graphics, о чем вы узнаете в главе 2 «Рисование с DirectX Graphics».

Чтобы изменить другие параметры окна, такие как высоту, ширину или тип, вы можете отредактировать находящиеся в начале кода определения констант. То же самое справедливо и для класса окна и его заголовка, которые объявлены в двух константах в начале кода.

Обратите внимание, что я добавил функцию, которая нигде не вызывается. Это функция **AppError**, которую я использую для отображения пользовательских сообщений об ошибках. Если ей в параметре **Fatal** передать значение **TRUE**, то работа приложения будет завершена, в то время как значение **FALSE** позволяет программе продолжать работу.

В каждой функции есть комментарии, объясняющие, что она делает, — ваша задача добавить код, выполняющий инициализацию объектов, загружающий графику, обрабатывающий кадры и т.д.

## Структурирование проекта

В начале каждого проекта для вас доступно множество вариантов. Различные функции, образующие ваше приложение, могут быть объединены в единый исходный файл или разделены по нескольким исходным файлам, согласно их функциональному назначению. Например, графические функции могут быть помещены в исходный файл графической подсистемы, звуковые — в исходный файл звуковой подсистемы и т.д. Затем вы включаете эти исходные файлы в ваш проект, предоставляете для каждого из них соответствующий заголовочный файл и больше ни о чем не волнуетесь.

Я всегда начинаю свои программы с файла `WinMain.cpp`. Он содержит точку входа приложения. Также в нем инициализируется окно и выполняются вызовы, необходимые для инициализации подсистем, выполняемых в каждом кадре функций и завершения работы (все эти функции могут находиться в отдельных исходных файлах). Фактически, подобный подход я использую во всей этой книге.

Глава 6, «Создание ядра игры», познакомит вас с набором классов базовых объектов, которые я использую для ускорения разработки своих игр. Вы можете включать эти файлы, разделенные по их функциональному назначению (графика, звук, сеть и т.д.) в ваши проекты (так же как и соответствующие им заголовочные файлы). Все, что вам потребуется потом сделать — это создать экземпляр класса и использовать этот объект по своему усмотрению.

Главная идея в следующем: разделяйте ваш проект на простые в использовании модули, которые не будут подавлять вас.

## Завершаем подготовку

Итак, теперь вы готовы к работе! Вы установили DirectX, настроили ваш компилятор для работы с ним и создали каркас приложения, который будет отправной точкой. Помимо этого, вы узнали как работать с потоками, критическими секциями, состояниями, процессами и пакетами данных. Надеюсь, это помогло вам!

Вооружившись приведенной в этой главе информацией, вы готовы погрузиться в мир программирования ролевых игр (да и любых других типов игр)! Примите мои поздравления и пожелания успеха!

### **Программы на CD-ROM**

Программы, демонстрирующие обсуждаемые в данной главе концепции расположены на прилагаемом к данной книге CD-ROM. Все они находятся в папке \BookCode\Chap01\:

**State** — демонстрация работы с описанным в главе стеком состояний. Местоположение: \BookCode\Chap01\State\.

**Process** — демонстрация использования описанного в данной главе стека процессов. Местоположение: \BookCode\Chap01\Process\.

**Data** — пример использования упаковки данных. Местоположение: \BookCode\Chap01\Data\.

**Shell** — отладочная консоль, используемая для создания окна и вызова различных функций, что облегчает разработку. Местоположение: \BookCode\Chap01\Shell\.

# **Часть II**

## **Основы DirectX**

**Глава 2      Рисование с DirectX Graphics**

**Глава 3      Взаимодействие с  
пользователем с DirectInput**

**Глава 4      Воспроизведение звуков и  
музыки с DirectX Audio и  
DirectShow**

**Глава 5      Работа с сетью с DirectPlay**

**Глава 6      Создаем ядро игры**





# Глава 2

## Рисование с DirectX Graphics

Современные игры поражают нас замечательной графикой и удивительными эффектами. Именно графические эффекты привлекают внимание большинства игроков, так что графика — это главный компонент в вашем проекте. К счастью, большинство графических библиотек и лежащие в их основе концепции достаточно прямолинейны и легки для понимания. Применяя базовые принципы рисования графики, вы сможете воссоздать те замечательные эффекты, которые видели в играх, а также создать ряд собственных новых эффектов.

Теперь пришло время заняться созданием собственной графической системы, и мы обратим внимание на DirectX Graphics — графическую компоненту DirectX. В этой главе я покажу вам использование DirectX Graphics, включая основные способы рисования и более сложные возможности DirectX Graphics, такие как наложение текстур и альфа-смешивание. К концу этой главы вы станете профессиональным программистом графики!

В данной главе я рассмотрю следующие темы:

- Графика в DirectX.
- Как работать в трех измерениях.
- Математика матриц.
- Использование библиотеки D3DX.
- Рисование с вершинами и полигонами.
- Работа с картами текстур.
- Использование альфа-смешивания.
- Щиты и частицы.
- Работа с сетками.
- Использование X-файлов.
- Анимация сеток.

## Сердце трехмерной графики

Хотя переход к такой сложной теме как трехмерная графика на данном этапе чтения книги может показаться нелогичным, в нем есть своя логика. Определенно, вся графическая система DirectX базируется на трехмерном прорыве Microsoft — DirectX3D. Поэтому все, что вы будете делать с DirectX Graphics, изложено в терминологии трехмерной графики.

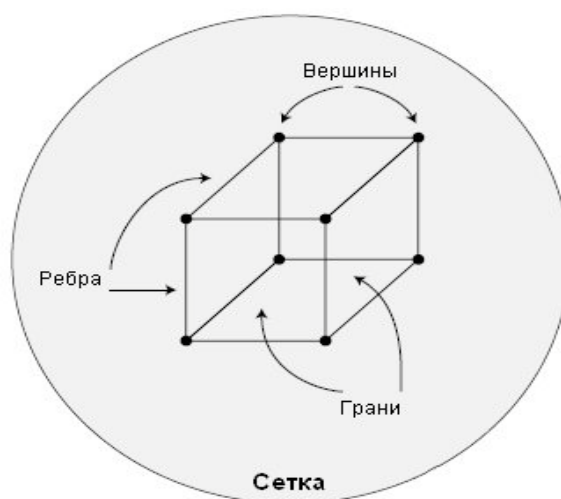
---

**ПРИМЕЧАНИЕ** В этом разделе я познакомлю вас с терминологией трехмерной графики и теорией рисования трехмерных изображений. Далее в этой главе, в разделе «Переходим к рисованию» вы примените теорию на практике, начав рисовать с использованием DirectX Graphics.

---

Все, и я действительно подразумеваю все, что рисуется с помощью DirectX3D состоит из полигонов. *Полигон (polygon)* — это обычно треугольная фигура, образованная тремя точками, называемыми *вершины*. *Вершина (vertex)* — это минимальная единица в трехмерной графике; единственная точка (с координатами) расположенная в двухмерном или трехмерном пространстве. Вы создаете ребра (линии), соединяя между собой две вершины, а три ребра образуют полигон. Вы можете думать об этих взаимосвязях между вершинами, ребрами и полигонами как об игре «соедини точки». Точки — это вершины, а линии, которые вы рисуете, — это ребра.

Три соединенных ребра образуют *грань (polygon face)*, а весь рисунок называется *сеткой (mesh)* или *моделью* (сетка — это результат объединения всех базовых объектов: вершин, ребер и полигонов). Взаимосвязь между вершинами, ребрами, полигонами и сетками показана на рис. 2.1.



**Рис. 2.1.** Соединяем точки для создания трехмерного объекта

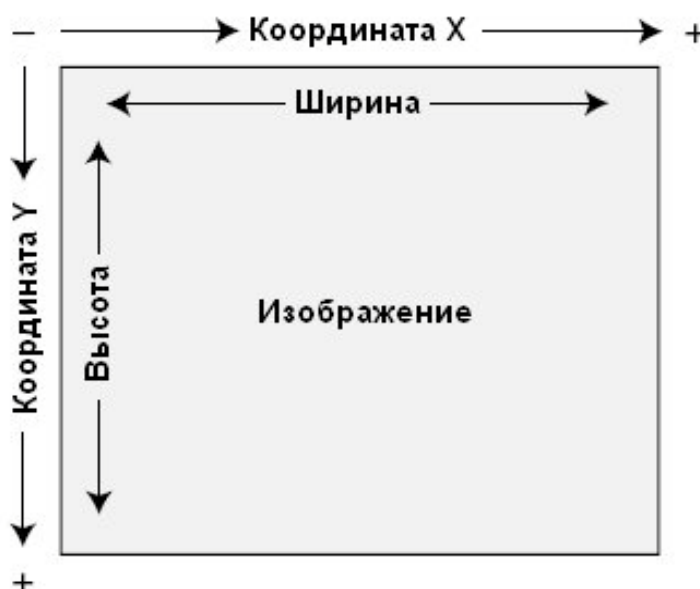
Чтобы представление объектов выглядело более реалистично, DirectX3D использует *материалы (materials)* для закрашивания пустых полигонов сетки указанным цветом. Материалы представляются в виде комбинации цветовых компонент, а также необязательных растровых изображений, называемых

текстуры (*texture*), которые в процессе визуализации наносятся на поверхность полигона.

## Системы координат

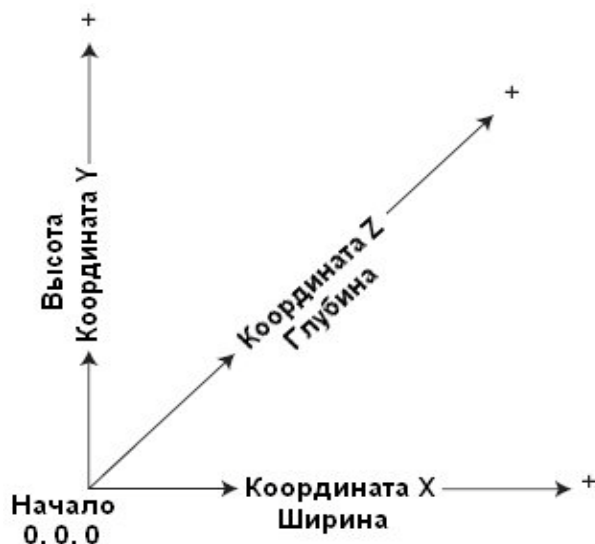
Скорее всего, вы уже знакомы с использованием двухмерной системы координат, если имели дело с изображениями, у которых есть ширина и высота, измеренные в пикселях. Горизонтальный отступ называют *координатой X*, а вертикальный отступ называют *координатой Y*. Каждая координата является смещением от верхнего левого угла изображения.

Система координат изображена на рис. 2.2. Координата X направлена слева направо от расположенного в крайней левой позиции начала координат. Координата Y направлена сверху вниз от расположенного на самом верху начала координат. Координаты увеличиваются в положительном направлении к другому концу их промежутков. В Direct3D двухмерные координаты обычно называют *преобразованными координатами* (*transformed coordinates*) потому что они представляют собой финальные координаты, используемые для рисования объекта на экране.



**Рис. 2.2.** Вы измеряете ширину и высоту изображения в пикселях. Вы указываете координаты в изображении используя соответствующие координаты X и Y

В трехмерном пространстве добавляется дополнительная координата — *координата Z*. Обычно координата Z представляет глубину изображения. Что более важно, координата Y переворачивается и направлена снизу вверх (положительное направление направлено вверх). Эта раскладка трех координат показана на рис. 2.3. Координата Z может использоваться двумя способами: когда положительное направление идет от зрителя или, когда положительное направление идет к зрителю. Эти два варианта обычно называются *левосторонней* (*left-handed*) и *правосторонней* (*right-handed*) координатными системами соответственно. В этой книге я использую левостороннюю систему координат.



**Рис. 2.3.** Трехмерные координаты:  $X$ ,  $Y$  и  $Z$ .  $X$  и  $Y$  аналогичны координатам на экране, а координата  $Z$  представляет глубину изображения

#### ПРИМЕЧАНИЕ

Левосторонняя и правосторонняя системы координат получили свои названия на основании способа определения направления третьей координаты с помощью руки. Разместите левую руку, чтобы ладонь была направлена вверх. Направьте большой палец от вас (остальные пальцы должны быть направлены вправо). Ваш большой палец указывает положительное направление оси  $Z$ , а остальные пальцы указывают положительное направление оси  $X$ . Теперь согните пальцы, чтобы они указывали вверх (не меняя положения руки), и они будут указывать положительное направление оси  $Y$ . Это справедливо для левосторонней системы координат. В случае с правосторонней системой координат все обстоит точно так же, но использовать надо правую руку.

Все в трехмерном мире измеряется в таких координатных системах — двухмерной для изображений и экрана и трехмерной для всего остального. Так, если вам требуется определить (с помощью трехмерных координат) точку пространства, находящуюся перед вами (по оси  $Z$ ), немного правее вас (по оси  $X$ ) и на уровне глаз (по оси  $Y$ ), вы можете задать для нее координаты  $X = 100$ ,  $Y = 50$ ,  $Z = 200$ . Эти координаты представляют точку, находящуюся в 100 единицах справа от вас, в 50 единицах над землей и в 200 единицах перед вами соответственно.

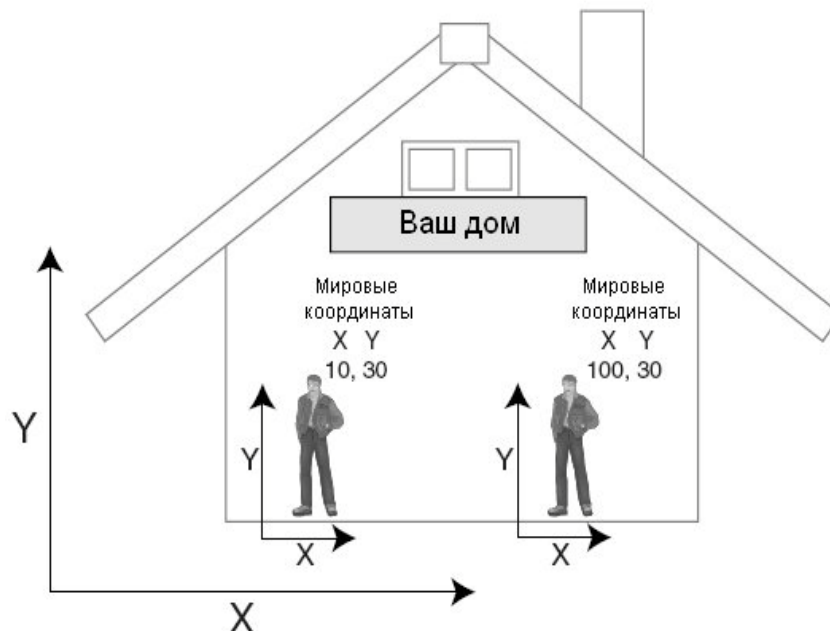
В случае двухмерных координат вы можете сказать, что картина на стене имеет 200 пикселей в ширину и 200 пикселей в высоту. Центр этой картины будет в точке  $X = 100$ ,  $Y = 100$ , а верхний левый угол — в точке  $X = 0$ ,  $Y = 0$ . Трехмерные координаты называют *непреобразованными координатами* (*untransformed coordinates*), поскольку они не представляют итоговые координаты, которые будут использоваться для визуализации объекта на дисплее. С другой стороны, двухмерные координаты называют *преобразованными координатами* (*transformed coordinates*), поскольку они непосредственно отображаются в координаты точек экрана. Позднее в этой

главе, в разделе «Трёхмерная математика», вы увидите как преобразовать непреобразованные координаты в преобразованные, а сейчас давайте сосредоточимся на том, как определить объекты, используя координаты о которых мы только что прочитали.

## Конструирование объектов

Конструируя объекты, такие как сетки и модели (и даже плоские двухмерные изображения), вы начинаете с уровня вершин. У каждой вершины есть назначенные ей координаты  $X$ ,  $Y$  и  $Z$ . Вы можете задать эти координаты тремя способами: в *экранном пространстве* (*screen space*) используя преобразованные координаты, в *пространстве модели* (*model space*) используя непреобразованные координаты, и в *мировом пространстве* (*world space*) также используя непреобразованные координаты.

Экранное пространство вы используете чтобы отобразить вершины на реальные координаты точек экрана. Пространство модели (также называемое *локальным пространством* — *local space*) ссылается на координаты, измеряемые относительно произвольной точки отсчета, которой обычно является центр модели. Вершины в локальном пространстве принадлежат модели, и вы можете перемещать их, придавая объекту требуемую форму. Перед визуализацией объекта вы преобразуете находящиеся в локальном пространстве вершины в мировое пространство. При визуализации объекта вы преобразуете координаты мирового пространства в координаты экранного пространства.



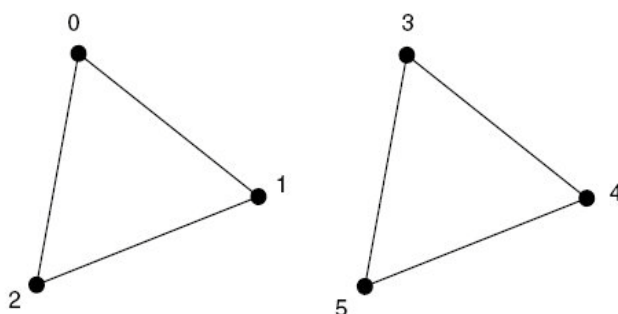
**Рис. 2.4.** Изредка при перемещении объекта в мире вы будете непосредственно манипулировать координатами вершин объекта. Чаще вместо этого вы указываете местоположение объекта, задавая его координаты в мировом пространстве, и позволяете Direct3D самому беспокоиться о размещении вершин

Вершины, размещенные в мировом пространстве представляют итоговое местоположение, используемое при визуализации объекта.

Мировое пространство — это действительная позиция относительно фиксированной точки в трехмерном мире. Для примера рассмотрим в качестве сетки вас. Ваши суставы — это вершины, определенные в локальном пространстве, поскольку их координаты заданы относительно центра вашей грудной клетки.

Когда вы перемещаетесь по дому (в мировом пространстве), координаты ваших суставов в мире меняются, но относительно центра вашего тела остаются постоянными, как это показано на рис. 2.4.

После выбора типа координат, используемых для рисования объекта (в экранном, локальном или мировом пространстве), вы размещаете вершины (нумеруя их в порядке их размещения). Затем вы объединяете эти вершины в группы по три для создания треугольных граней. На рис. 2.5 показан набор полигональных граней созданных путем группировки вершин.



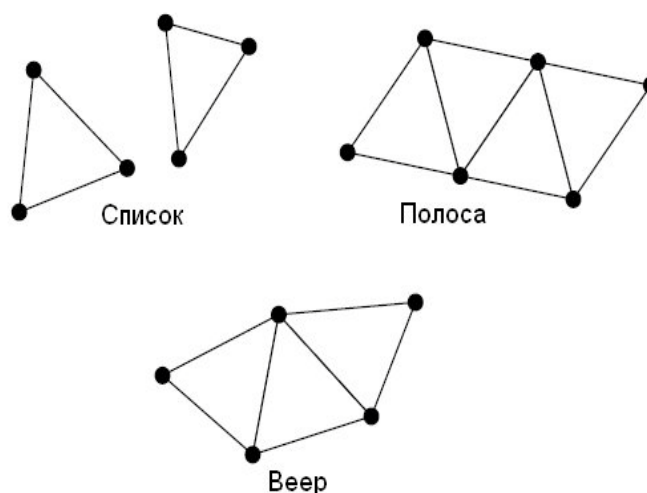
**Рис. 2.5.** Вы используете шесть вершин для рисования двух полигонов. Каждый полигон должен быть треугольником, поэтому каждый полигон использует только три вершины

## Списки, полосы и веера

При создании полигональных граней вы обязательно должны учесть совместное использование вершин (когда один полигон использует ту же самую вершину, или вершины, что и другой). Набор полигональных граней может попадать в одну из трех категорий: списки треугольников, полосы треугольников и веера треугольников.

*Список треугольников (triangle list)* — это набор граней не имеющих общих вершин, поэтому у каждого полигона свое трио вершин. *Полоса треугольников (triangle strip)* — это набор граней с общими вершинами в котором у каждого полигона общее ребро с другим. *Веер треугольников (triangle fan)* получается в том случае, если несколько граней совместно используют одну и ту же вершину, подобно соединенным в одной точке пластинам настоящего веера. Эти три категории показаны на рис. 2.6.

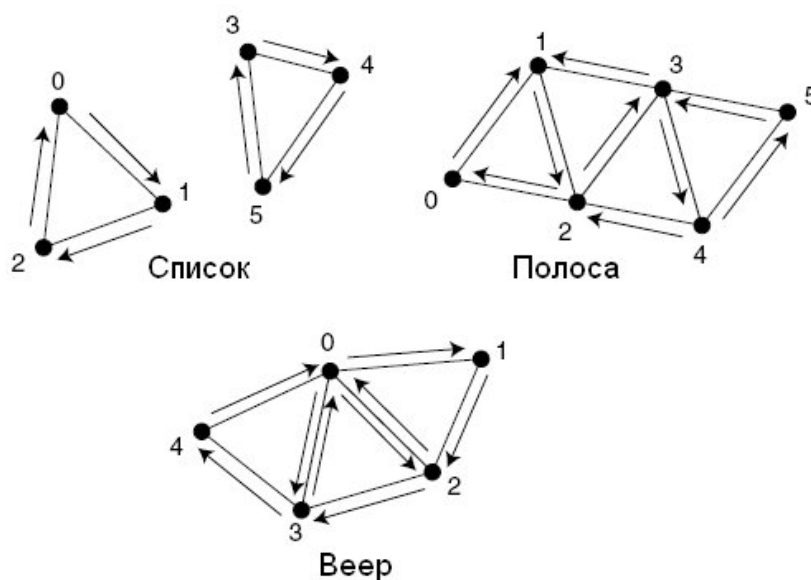
Помимо использования треугольных полигонов у вас есть возможность визуализировать отдельные пиксели и линии. В DirectX3D отдельные рисуемые пиксели называются *точками (point)*, а линии называются... да так и называются, *линиями (line)*. Для визуализации точек и линий используются соответственно одна и две вершины. Поскольку мы сосредотачиваемся на трехмерной графике, я пропущу обсуждение типов точек и линий и сосредоточусь исключительно на типах треугольников.



**Рис. 2.6.** В списке треугольников, в отличие от полос и вееров треугольников, вершины совместно не используются. Применение полос и вееров сокращает количество вершин, что экономит память и увеличивает скорость визуализации

## Порядок вершин

Позднее, когда вы переходите к визуализации полигонов, очень важен порядок, который использовался при определении вершин, потому что вы должны определить, какая сторона грани является лицевой, а какая — обратной. Для текущих целей вы будете упорядочивать вершины, которые определяют грань так, чтобы они шли по часовой стрелке (когда вы смотрите на лицевую сторону полигона), как показано на рис. 2.7. Таким образом, если образующие грань вершины идут по часовой стрелке, то вы знаете, что смотрите на лицевую поверхность грани.



**Рис. 2.7.** Обратите внимание на порядок вершин, поскольку при конструировании списков, полос и вееров для вершин используются особые порядки вершин



**ПРИМЕЧАНИЕ**

В системах трехмерной графики обращенные к зрителю обратной стороной грани обычно не рисуются и игнорируются в процессе визуализации. Этот процесс называется *отбрасыванием обратных граней* (*backface culling*) и является одним из основных способов оптимизации. В правосторонней системе координат все выглядит с точностью до наоборот, и лицевой стороной к зрителю обращены те грани, у которых вершины идут против часовой стрелки.

---

Проницательные читатели вероятно заметили, что в полосе треугольников на рис. 2.7, для каждого следующего треугольника порядок вершин меняется на противоположный. Это изменение порядка вершин необходимо для рисования полос треугольников в Direct3D.

## Окрашивание полигонов

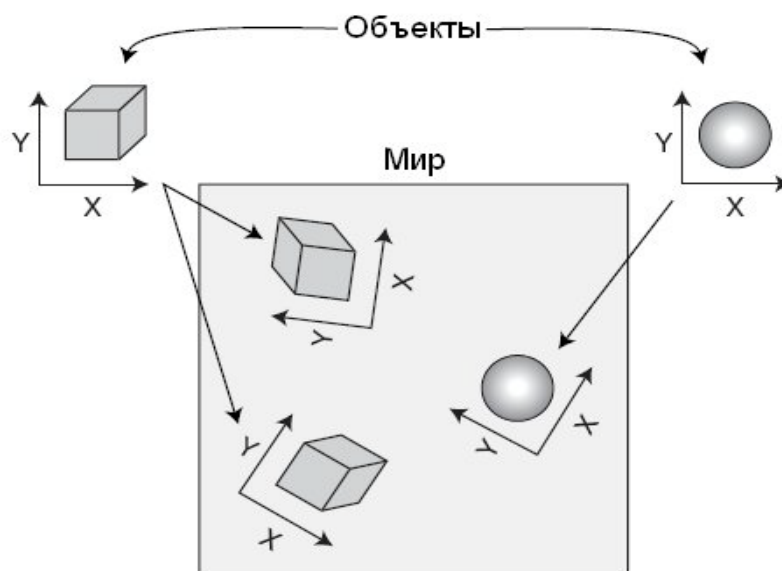
После того, как вы определили группы полигонов или сетку, вы готовы раскрашивать грани. Direct3D предлагает две простых техники, которые я буду обсуждать в этой книге. Первая техника использует определения *материалов* (*material*), которые фактически являются простыми цветами. Для материалов задаются их рассеиваемая, фоновая и отражаемая цветовые компоненты. *Рассеиваемый* (*diffuse*) и *фоновый* (*ambient*) цвета обычно один и тот же цвет, — тот цвет в который действительно окрашен объект. *Отражаемый* (*specular*) цвет — это цвет бликов, появляющихся на объекте, если осветить его близко расположенным источником света. (Подробнее об этих трех цветовых компонентах мы поговорим позже в разделе «Материалы и цвета» этой главы.)

Вторая техника, называемая *наложением текстур* (*texture mapping*), раскрашивает полигоны используя изображение. Сама текстура является изображением, обычно загружаемым из файла с растровой графикой. Это растровое изображение натягивается или накладывается повторяющейся мозаикой на поверхность полигона.

## Преобразования

После того, как вы определили модель (или даже просто набор полигонов), можно разместить ее в требуемом месте мира. На рис. 2.8 показаны несколько моделей, которые помещаются в трехмерный мир. Вы можете перемещать, масштабировать и вращать любой объект как вам надо, так что можно использовать одну модель для рисования нескольких объектов с различной ориентацией.

Такие действия, как передвижение (также называемое *перемещением*, *translating*), вращение и масштабирование называются *преобразованиями* (*transformation*). Для того, чтобы перенести объект из его пространства модели в готовую к просмотру систему координат необходим набор преобразований.



**Рис. 2.8.** Хотя вы определяете трехмерные объекты в их собственных локальных пространствах, потом вы должны разместить их в мировом пространстве

Первое из них, *мировое преобразование* (*world transformation*), применяется для преобразования объекта из его локальных координат в мировые координаты. Это включает масштабирование, вращение относительно осей  $X$ ,  $Y$  и  $Z$ , и перемещение (именно в этом порядке). Следующее преобразование — это *преобразование вида* (*view transformation*), которое ориентирует все объекты относительно точки просмотра в трехмерном мире, то есть преобразует мировые координаты в координаты вида.

О преобразовании вида можно думать как о камере, летающей в вашем трехмерном мире. Свободно перемещающаяся и вращающаяся камера является порталом для просмотра вашего виртуального мира. Преобразование вида перемещает объекты вашего мира в систему координат, центром которой является камера.

Последнее важное преобразование — это *преобразование проекции* (*projection transformation*), которое преобразует ваш трехмерный мир в его двухмерное изображение. Оно похоже на объектив камеры с изменяемым фокусным расстоянием, настраиваемыми углами обзора и различными эффектами, такими как сверхширокоугольный обзор.

## Начинаем работать с DirectX Graphics

Теперь, когда вы познакомились с основами рисования трехмерной графики, пришло время использовать полученные знания на практике. Однако перед этим вам необходимо узнать, как подготовить к работе графическую систему.

В данном разделе я познакомлю вас с компонентами DirectX Graphics, которые будут использоваться в этой книге и покажу как запустить графическую систему и подготовить ее к рисованию.

---

**ПРИМЕЧАНИЕ** Хотя в последних версиях графические компоненты и называются DirectX Graphics, я буду ссылаться как на Direct3D, поскольку его используют все трехмерные графические объекты.

---



---

**ПРИМЕЧАНИЕ** Чтобы в вашем проекте использовать Direct3D надо добавить в него заголовочный файл D3D9.H и библиотеку D3D9.LIB.

---

## Компоненты Direct3D

Direct3D разделяет графический функционал по нескольким COM-объектам. У каждого объекта есть собственное назначение, например объект **IDirect3D9** используется для управления всей графической системой в целом, а объект **IDirect3DDevice9** применяется для управления процессом визуализации графики на экране.

В этой книге я покажу вам только те объекты, которые перечислены в таблице 2.1; именно эти объекты вы скорее всего будете использовать в вашем проекте игры.

**Таблица 2.1.** Основные компоненты Direct3D

<i>Компонент</i>	<i>Описание</i>
<b>IDirect3D9</b>	Используйте этот объект для сбора информации об установленном графическом оборудовании и инициализации интерфейсов устройств
<b>IDirect3DDevice9</b>	Имеет дело непосредственно с аппаратурой трехмерной графики. С ним вы визуализируете графику, управляете ресурсами изображений, создаете и устанавливаете режимы визуализации, фильтры затенения и т.д.
<b>IDirect3DVertexBuffer9</b>	Содержит массив данных вершин, используемый для рисования полигонов
<b>IDirect3DTexture9</b>	Используйте этот объект для хранения всех изображений, применяемых для рисования граней в трехмерных (и двухмерных) изображениях

---

**ПРИМЕЧАНИЕ** Хотя есть еще несколько компонентов Direct3D, которые можно использовать, они выходят за рамки данной книги. Чтобы больше узнать об этих дополнительных объектах, обратитесь к документации DirectX SDK.

---

## Инициализация системы

Благодаря упрощению Direct3D, начало использования графической системы это простая задача. Вот четыре общих этапа инициализации и запуска графической системы:

1. Получение интерфейса Direct3D.
2. Выбор видеорежима.
3. Установка метода показа.
4. Создание интерфейса устройства и инициализация дисплея.

Это очень короткий список! Я же сказал вам, что инициализировать и запустить графическую систему очень просто, так что давайте двигаться дальше и посмотрим как выполняется каждый из этих этапов.

## Получение интерфейса Direct3D

Первый шаг к использованию графики — инициализация объекта **IDirect3D9**. Выполняется она с помощью функции **Direct3DCreate9**.

```
IDirect3D9 *Direct3DCreate9(UINT SDKVersion);    // D3D_SDK_VERSION
```

---

**ПРИМЕЧАНИЕ** Большинство функций DirectX (также как все COM-объекты) возвращают значение типа **HRESULT**. Время от времени вы будете встречать функции (такие, как **Direct3DCreate9**), которые возвращают значение типа отличного от **HRESULT**, так что будьте начеку.

---

Единственным аргументом этой функции должна быть константа **D3D\_SDK\_VERSION**, указывающая используемую версию SDK. Возвращаемая переменная — это указатель на созданный для вас объект **IDirect3D9**, или, если при создании объекта Direct3D произошла ошибка, будет возвращен **NULL**.

Для использования этой функции достаточно объявить экземпляр объекта **IDirect3D9** и вызвать саму функцию:

```
IDirect3D9 g_D3D; // Глобальный объект IDirect3D9

if((g_D3D = Direct3DCreate9(D3D_SDK_VERSION)) == NULL) {
    // Произошла ошибка
}
```

## Выбор видеорежима

После того, как объект **IDirect3D9** создан, можно использовать его для получения информации о графической системе, которая включает список видеорежимов, которые может поддерживать Direct3D. Фактически, если вы не хотите менять формат, то можете запросить у объекта **IDirect3D9** информацию о текущем видеорежиме.

Видеорежимы различаются по разрешению экрана (высоте и ширине в пикселях), глубине цвета (количеству отображаемых цветов) и частоте кадров. Например, вы можете установить разрешение  $640 \times 480$  точек с 16-разрядной глубиной цвета и предлагаемой видеокартой по умолчанию частотой кадров.

Информация о видеорежиме хранится в структуре **D3DDISPLAYMODE**.

```
typedef struct _D3DDISPLAYMODE {
    UINT      Width;          // Ширина экрана в пикселях
    UINT      Height;         // Высота экрана в пикселях
    UINT      RefreshRate;    // Частота кадров (0 = по умолчанию)
    D3DFORMAT Format;          // Формат цвета
} D3DDISPLAYMODE;
```

Относительно высоты, ширины и частоты кадров все ясно, но что насчет формата цвета? В графике вы обычно выбираете сколько бит будет использоваться в одном пикселе (16, 24 или 32) для хранения информации о цвете. Чем больше битов используется, тем больше цветов можно отобразить (и тем больше потребуется памяти).

Цветовые режимы обычно различаются по количеству битов, отводимых для каждого компонента цвета (красного, зеленого, синего и, возможно, альфа). Возьмем к примеру 16-разрядный цветовой режим — 5 бит для красного, 5 бит для зеленого, 5 бит для синего и один бит для альфа-канала. 5 бит для хранения позволяют использовать 32 оттенка каждого компонента. Для альфа-канала используется один бит, который может быть либо установлен, либо сброшен.

Когда вы ссылаетесь на цветовой режим, вам недостаточно только сказать, что он 16-разрядный, а надо указать количество битов для компонентов цвета, например 1555 (1 для альфа, 5 для красного, 5 для зеленого и 5 для синего). Стандартными цветовыми режимами являются 555 (5 для красного, 5 для зеленого, 5 для синего без альфа-канала), 565 (5 для красного, 6 для зеленого и 5 для синего) и 888 (по 8 бит для каждого компонента). Заметьте, что значение альфа нужно не всегда.

В Direct3D цветовые режимы определены как члены перечисления, представленные в таблице 2.2.

**Таблица 2.2.** Константы цветовых режимов в Direct3D

<i><b>Значение</b></i>	<i><b>Формат</b></i>	<i><b>Описание</b></i>
<b>D3DFMT_R8G8B8</b>	24 разряда	8 красный, 8 зеленый, 8 синий
<b>D3DFMT_A8R8G8B8</b>	32 разряда	8 альфа, 8 красный, 8 зеленый, 8 синий
<b>D3DFMT_X8R8G8B8</b>	32 разряда	8 не используется, 8 красный, 8 зеленый, 8 синий
<b>D3DFMT_R5G6B5</b>	16 разрядов	5 красный, 6 зеленый, 5 синий
<b>D3DFMT_X1R5G5B5</b>	16 разрядов	1 не используется, 5 красный, 5 зеленый, 5 синий
<b>D3DFMT_A1R5G5B5</b>	16 разрядов	1 альфа, 5 красный, 5 зеленый, 5 синий

На данном этапе предположим, что вы хотите установить видеорежим  $640 \times 480$  и использовать формат цвета **D3DFMT\_R5G6B5**. Вот как в этом случае должна выглядеть инициализация структуры **D3DDISPLAYMODE**:

```
D3DDISPLAYMODE d3ddm;

d3ddm.Width      = 640;
d3ddm.Height     = 480;
d3ddm.RefreshRate = 0; // по умолчанию
d3ddm.Format      = D3DFMT_R5G6B5;
```

Чтобы проверить поддерживает ли видеокарта необходимый вам формат цвета, заполните структуру **D3DDISPLAYFORMAT** информацией о видео режиме и выполните вызов функции:

```
// g_pD3D = ранее инициализированный объект Direct3D
// d3df    = ранее инициализированная структура D3DFORMAT
// Проверка наличия видеорежима
if (FAILED(g_pD3D->CheckDeviceType(D3DADAPTER_DEFAULT,
                                   D3DDEVTYPE_HAL, &d3df, &d3df, FALSE))) {
    // Ошибка - цветовой режим не поддерживается
}
```

Задание видеорежима подразумевает, что вы работаете в полноэкранном режиме. Если же вы хотите использовать оконный режим (также, как и обычные приложения Windows), Direct3D может предоставить вам информацию об установленном видеорежиме. Для этого предназначен следующий вызов функции:

```
// g_pD3D = ранее инициализированный объект Direct3D
D3DDISPLAYMODE d3ddm;

if (FAILED(g_pD3D->GetDisplayMode(D3DADAPTER_DEFAULT, &d3ddm))) {
    // Произошла ошибка
}
```

#### ПРИМЕЧАНИЕ

Так же как и все COM-интерфейсы, Direct3D возвращает значение типа **HRESULT**. Значение **D3D\_OK** сообщает, что работа функции завершена успешно; любое другое значение свидетельствует об ошибке. Для простой проверки возвращаемых кодов вы можете использовать стандартные макросы **FAILED** или **SUCCEEDED**.

В случае успешного завершения показанный выше вызов **IDirect3D9::GetDisplayMode** возвращает заполненную данными структуру **D3DDISPLAYMODE**.

#### ВНИМАНИЕ!

Некоторые видеокарты не позволяют использовать определенные видеорежимы. Вы должны сами определить, какие видеорежимы поддерживаются видеокартой. Если ваша программа будет работать в оконном режиме, это не является проблемой, поскольку Direct3D обрабатывает параметры цветового режима за вас.

## Установка метода показа

Следующий этап подготовки Direct3D — определение того, как графика будет показываться пользователю. Вы хотите использовать окно, полный экран или вторичный буфер (о вторичном буфере вы можете прочитать в приведенном ниже примечании)? Какую частоту кадров вы хотите использовать? Вся эта информация (и, как вы увидите, еще и некоторая другая) хранится в структуре **D3DPRESENT\_PARAMETERS**:

```
typedef struct _D3DPRESENT_PARAMETERS
{
    UINT          BackBufferWidth;    // Ширина вторичного буфера
    UINT          BackBufferHeight;  // Высота вторичного буфера
    D3DFORMAT     BackBufferFormat;  // Формат экрана
    UINT          BackBufferCount;    // 1

    D3DMULTISAMPLE_TYPE MultiSampleType; // 0
    D3DSWAPEFFECT     SwapEffect;        // Способ показа
                                           // вторичного буфера

    HWND hDeviceWindow; // NULL
    BOOL Windowed;      // TRUE для оконного режима
                       // FALSE для полноэкранного режима

    BOOL          EnableAutoDepthStencil; // FALSE
    D3DFORMAT     AutoDepthStencilFormat; // 0

    DWORD         Flags; // 0

    UINT          FullScreen_RefreshRateInHz; // 0
    UINT          PresentationInterval;       // 0
} D3DPRESENT_PARAMETERS;
```

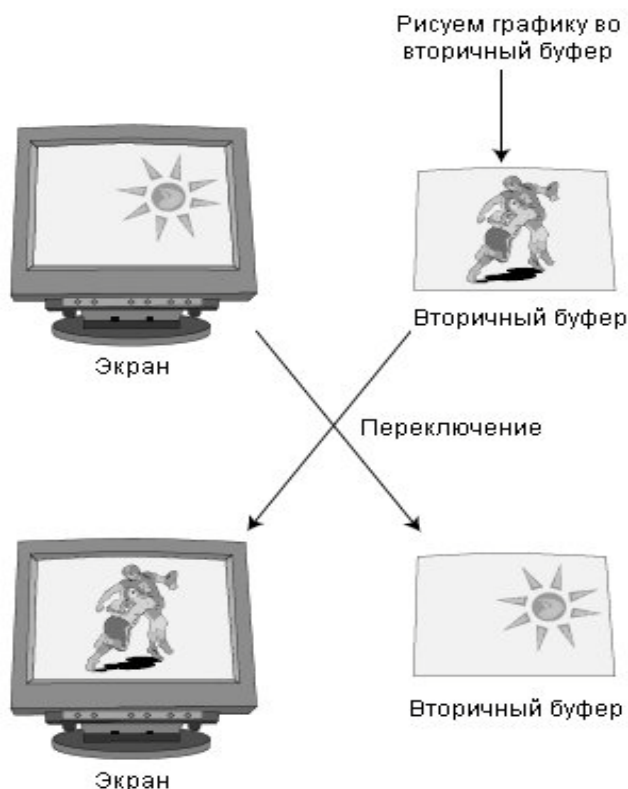
---

**ПРИМЕЧАНИЕ** *Вторичный буфер (back buffer) — это внеэкранная графическая поверхность (того же размера, что и окно или экран), которая получает все результаты операций рисования. Чтобы увидеть графику, помещенную во вторичный буфер, вы используете операцию, называемую **переключение (flip)**, которая отображает содержимое вторичного буфера на экране или в окне. Эта операция обеспечивает плавное обновление — пользователь никогда не видит, что рисует программа, пока изображение не будет полностью готово для вывода на экран.*

*Эта концепция иллюстрируется на рис. 2.9, где показаны первичный буфер (экран) и вторичный буфер (внеэкранная поверхность). Вы рисуете во вторичном буфере и, когда закончите рисование, выполняете переключение, чтобы вторичный буфер был отображен на экране.*

---

Хотя данная операция может показаться сложной, вам в действительности не придется иметь дела с большинством полей структуры **D3DPRESENT\_PARAMETERS**; однако вы должны понимать назначение полей, связанных с вторичным буфером.



**Рис. 2.9.** Рисование во вторичном буфере скрывает от вас объекты, пока вы не выполните переключение экранов

Вот два варианта инициализации, которые вы можете использовать в зависимости от того, в полноэкранном или в оконном режиме вы собираетесь работать (с небольшим фрагментом, который используется в обоих вариантах):

```
// d3ddm = ранее инициализированная структура D3DDISPLAYMODE
D3DPRESENT_PARAMETERS d3dpp;

// Очищаем структуру
ZeroMemory(&d3dpp, sizeof(D3DPRESENT_PARAMETERS));

// Для оконного режима используйте:
d3dpp.Windowed          = TRUE;
d3dpp.SwapEffect         = D3DSWAPEFFECT_DISCARD;
d3dpp.BackBufferFormat   = d3ddm.Format; // Используем тот же цветовой
                                         // режим

// Для полноэкранного режима используйте:
d3dpp.Windowed          = FALSE;
d3dpp.SwapEffect         = D3DSWAPEFFECT_FLIP;
d3dpp.FullScreen_RefreshRateInHz = D3DPRESENT_RATE_DEFAULT;
d3dpp.PresentationInterval = D3DPRESENT_INTERVAL_DEFAULT;
d3dpp.BackBufferFormat   = d3ddm.Format; // используем тот же
                                         // цветовой режим

// И для оконного и для полноэкранного режима указываем ширину и высоту
d3dpp.BackBufferWidth    = Width; // Укажите ваше значение ширины
d3dpp.BackBufferHeight   = Height; // Укажите ваше значение высоты
```



## Создание интерфейса устройства и инициализация дисплея

Наконец вы можете создать интерфейс устройства Direct3D — рабочую лошадку системы трехмерной графики. Чтобы создать и инициализировать интерфейс дисплея вызовите функцию **IDirect3D9::CreateDevice**, передав ей ранее инициализированные экземпляры структур **D3DDISPLAYMODE** и **D3DPRESENT\_PARAMETERS**:

```
HRESULT IDirect3D9::CreateDevice(
    UINT          Adapter,          // D3DADAPTER_DEFAULT
    D3DDEVTYPE DeviceType,         // D3DDEVTYPE_HAL
    HWND          hFocusWindow,    // Deskриптор окна для визуализации
    DWORD         BehaviorFlags,    // D3DCREATE_SOFTWARE_VERTEXPROCESSING
    D3DPRESENT_PARAMETERS *pPresentationParameters, // d3dpp
    IDirect3DDevice9 *ppReturnedDeviceInterface); // Объект устройства
```

У функции **CreateDevice** есть параметры для передачи структуры параметров показа, которую вы создали, а также дескриптора окна, принадлежащего вашему приложению (которое Direct3D будет использовать для отображения визуализированной графики). Остальные аргументы достаточно стандартны и вам редко придется менять их значения. Последний аргумент — это указатель на объект устройства Direct3D, который вы создадите. Вызов **IDirect3D9::CreateDevice** может выглядеть так:

```
// g_pD3D = ранее инициализированный объект Direct3D
// hWnd   = дескриптор используемого для визуализации окна
// d3dpp   = ранее инициализированная структура параметров показа
IDirect3DDevice9 *g_pD3DDevice;
if(FAILED(g_pD3D->CreateDevice(D3DADAPTER_DEFAULT,
                               D3DDEVTYPE_HAL, hWnd,
                               D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                               &d3dpp, &g_pD3DDevice))) {

    // Произошла ошибка
}
```

### ПРИМЕЧАНИЕ

Direct3D работает с двумя различными типами устройств — HAL (Hardware Abstraction Layer) и REF (REFeence device). Устройство HAL — это установленная у вас видеокарта, и именно это устройство вы будете использовать в большинстве случаев при разработке игр. Устройство REF (объект программного визуализатора) используется когда вам надо отладить приложение или проверить возможность, которая не поддерживается установленной видеокартой. HAL более быстрое из устройств, поскольку позволяет Direct3D работать непосредственно с аппаратурой видеокарты, а устройство REF выполняет всю работу программно (и поэтому работает исключительно медленно).

## Потеря устройства

Обычно интерфейс устройства функционирует как ожидается и все замечательно работает; графика рисуется и выполняется управление ресурсами памяти. Хотя было бы хорошо думать, что дела будут всегда обстоять подобным образом, это не так. Добро пожаловать в мир потерянных устройств.

*Потерей устройства (lost device)* называется утрата контроля над графическими ресурсами по какой-либо причине. Может быть, другое приложение получило контроль над видеокартой и освободило память в которой хранились графические ресурсы вашей программы. Возможно, Windows остановила приложение, переведя систему в спящий режим. Независимо от причины вы утратили контроль над видеокартой и вам необходимо вернуть его.

Как узнать что контроль утерян? Проверив значение, возвращаемое любой функцией объекта устройства! Например, далее в этой главе в разделе «Показ сцены», вы увидите как отобразить графику на экране монитора. Если этот вызов возвращает значение **D3DERR\_DEVICELOST**, вы знаете, что устройство потеряно.

Восстановление контроля над устройством это, можно сказать, решительный поступок. Выполняется он с помощью следующей функции:

```
HRESULT IDirect3DDevice9::Reset(  
    D3DPRESENT_PARAMETERS *pPresentationParameters);
```

Единственным параметром функции является структура параметров показа, которую вы использовали при инициализации устройства:

```
// g_pD3DDevice = ранее инициализированный объект устройства  
// d3dpp        = ранее инициализированная структура параметров показа  
g_pD3DDevice->Reset(&d3dpp);
```

Я был бы рад сказать, что это волшебная функция, которая сделает за вас все, что необходимо для восстановления устройства, но, к сожалению вынужден сообщить плохие новости. Вызов этой функции выполняет сброс устройства, что приводит к удалению всех ресурсов — на самом деле это не так уж и плохо, потому что есть шанс, что эти ресурсы уже были потеряны (из-за потери устройства).

В результате вы должны заново загрузить все графические ресурсы (такие как текстуры) и восстановить состояние устройства (режимы визуализации). Большинство потерянного — это данные о которых мы еще не говорили, поэтому сейчас мы завершим обсуждение темы.

## Знакомство с D3DX

Работа с Direct3D остается главной задачей. Хотя Microsoft упростила многие интерфейсы, для вас остается много различной работы. Чтобы сократить время разработки приложений Microsoft создала библиотеку D3DX. Библиотека D3DX объединяет различные относящиеся к графике

полезные функции, такие как работа с сетками, текстурами, шрифтами, математикой и т.д. В этой книге вы увидите как можно использовать библиотеку D3DX чтобы облегчить процесс разработки игр.

---

**ПРИМЕЧАНИЕ** Все функции библиотеки D3DX начинаются с префикса **D3DX** (например, **D3DXCreateFont**). Библиотека D3DX содержит не только функции, но и COM-объекты, такие как **ID3DXBaseMesh**.

---

---

**ПРИМЕЧАНИЕ** Чтобы использовать библиотеку D3DX в своем проекте вы должны включить в него заголовочный файл D3DX9.H и компоновать его с библиотекой D3DX9.LIB. Кроме того, вы можете добавить ссылку на библиотеку D3DXOF.LIB, которая потребуется вам в дальнейшем.

---

## Трехмерная математика

Как вы, возможно, уже поняли, использование трехмерной графики сопряжено с соответствующей математикой и имеет дело с таким количеством чисел, что в них легко запутаться. Некоторое время назад трехмерная графика в реальном времени было скорее мечтой, чем реальностью. Компьютеры просто не могли выполнять вычисления с необходимой скоростью.

Конечно, со временем дела шли лучше, и сейчас мы в состоянии реализовать сногшибательные эффекты. Одна из причин такого изменения — достижения в математике, связанной с трехмерной графикой.

## Матричная алгебра

Нет, этот раздел не посвящен Киану Ривзу и его следующему фильму, в котором он застревает внутри калькулятора. *Матричная алгебра* (*matrix math*) — это раздел линейной алгебры помогающий упростить и сократить отдельные вычисления. В данном случае это вычисления трехмерных преобразований, о которых я уже упоминал.

Поскольку каждый трехмерный объект состоит из множества вершин, работа Direct3D заключается в преобразовании этих вершин в координаты, пригодные для визуализации графики на экране. Чтобы создать сцену в каждом кадре вы должны преобразовать тысячи вершин. Это достаточно серьезная математика — достаточно чтобы перехватило дух у любого преподавателя математики.

---

**ВНИМАНИЕ!** Обратите внимание, что матричная алгебра используется только когда вы работаете с координатами в трехмерном пространстве. Если вы используете преобразованные координаты (координаты в экранном пространстве), вам не надо применять к ним никаких дополнительных преобразований.

---

Direct3D работает с преобразованиями с помощью матриц. *Матрица* (*matrix*) — это таблица чисел в которой каждый элемент имеет определенное назначение. В рассматриваемом сейчас случае числа представляют преобразования, которые вы хотите применить к вершине. Комбинируя все необходимые вычисления в упакованную форму, такую как матрица, вы значительно упрощаете математику.

## Конструирование матриц

Матрицы могут быть любого размера, но сейчас мы сосредоточимся на матрицах размера  $4 \times 4$ , то есть такие у которых четыре строки и четыре столбца. Direct3D хранит матрицы в виде структуры **D3DMATRIX**:

```
typedef struct _D3DMATRIX {
    D3DVALUE _11, _12, _13, _14;
    D3DVALUE _21, _22, _23, _24;
    D3DVALUE _31, _32, _33, _34;
    D3DVALUE _41, _42, _43, _44;
} D3DMATRIX;
```

Чтобы заполнить матрицу данными преобразования, которое вы хотите использовать, можно воспользоваться библиотекой D3DX. Вместо использования структуры **D3DMATRIX** воспользуйтесь объектом **D3DXMATRIX**, который содержит те же переменные, что и **D3DMATRIX** и плюс к этому еще ряд полезных функций.

Каждое преобразование, которое вы будете использовать, имеет собственную матрицу, за исключением вращения для которого необходимы три матрицы (по одной для каждой оси). Это означает, что вам необходимы пять матриц преобразования: вращение относительно оси X, вращение относительно оси Y, вращение относительно оси Z, перемещение и масштабирование. Первый набор функций используется для инициализации матриц вращения:

```
D3DXMATRIX *D3DXMatrixRotationX(
    D3DXMATRIX *pOut, // Итоговая матрица
    FLOAT Angle);     // Угол поворота вокруг оси X

D3DXMATRIX *D3DXMatrixRotationY(
    D3DXMATRIX *pOut, // Итоговая матрица
    FLOAT Angle);     // Угол поворота вокруг оси Y

D3DXMATRIX *D3DXMatrixRotationZ(
    D3DXMATRIX *pOut, // Итоговая матрица
    FLOAT Angle);     // Угол поворота вокруг оси Z
```

### ПРИМЕЧАНИЕ

Обратите внимание, что все функции работы с матрицами в библиотеке D3DX также возвращают указатель на **D3DXMATRIX**. Это указатель на итоговую матрицу и его наличие позволяет использовать функции работы с матрицами в выражениях, как в приведенном ниже примере:

```
D3DXMATRIX matMatrix, matResult;
matResult = D3DXMatrixRotationZ(&matMatrix, 1.57f);
```

Передав каждой из перечисленных выше функций пустую матрицу и значение угла поворота (в радианах, представляющее угол поворота относительно соответствующей оси), вы получите матрицу, заполненную необходимыми для работы значениями.

Следующая функция создает матрицу перемещения, которая применяется для передвижения объектов:

```
D3DXMATRIX *D3DXMatrixTranslation(  
    D3DXMATRIX *pOut, // Итоговая матрица  
    FLOAT x,           // Координата по оси X  
    FLOAT y,           // Координата по оси Y  
    FLOAT z);          // Координата по оси Z
```

Координаты — это фактические смещения объекта относительно начала системы координат. Перемещение используется для преобразования объекта из координат его локального пространства в мировые координаты. Теперь представим функцию, которая масштабирует объект по его осям:

```
D3DXMATRIX *D3DXMatrixScaling(  
    D3DXMATRIX *pOut, // Итоговая матрица  
    FLOAT sx,         // Масштаб по X  
    FLOAT sy,         // Масштаб по Y  
    FLOAT sz);        // Масштаб по Z
```

Обычный масштаб объекта равен 1.0. Чтобы увеличить размер объекта в два раза, укажите значение 2.0, а чтобы сделать объект в половину меньше его оригинального размера, задайте значение 0.5. Вы также будете использовать специальный тип матрицы, называемый *единичной матрицей* (*identity matrix*). Большая часть элементов такой матрицы равна нулю, а некоторые равны единице. Будучи примененной к другой матрице, единичная матрица не оказывает никакого эффекта и оставляет значения элементов результата теми же самыми, что и в оригинале. Единичная матрица очень полезна, когда вы хотите скомбинировать две матрицы, но не желаете изменять оригиналы.

Для создания единичной матрицы можете использовать следующую функцию (которая получает в своем единственном параметре матрицу):

```
D3DXMATRIX *D3DXMatrixIdentity(D3DXMATRIX *pOut);
```

Хотя прототипы функций не очень сложны, вот несколько примеров:

```
D3DXMATRIX matXRot, matYRot, matZRot;  
D3DXMATRIX matTrans, matScale;  
  
// Задаем поворот на 45 градусов (.785 радиан)  
D3DXMatrixRotationX(&matXRot, 0.785f);  
D3DXMatrixRotationY(&matYRot, 0.785f);  
D3DXMatrixRotationZ(&matZRot, 0.785f);  
  
// Задаем перемещение в точку 100,200,300  
D3DXMatrixTranslation(&matTrans, 100.0f, 200.0f, 300.0f);  
  
// Увеличиваем размер объекта в два раза по всем осям  
D3DXMatrixScaling(&matScale, 2.0f, 2.0f, 2.0f);
```

## Комбинирование матриц

После заполнения различных матриц значениями, используемыми для преобразований, вы можете применить их к каждой вершине. Фактически, вы можете сделать еще проще, скомбинировать отдельные матрицы, содержащие значения для перемещения, вращения и масштабирования в единую матрицу, перемножив их между собой. Эта процедура называется *конкатенацией матриц* (*matrix concatenation*), и является основой оптимизации всех матричных вычислений.

Создав единую матрицу для кадра, вы можете использовать ее для каждой вершины в сцене. Применение этой единой матрицы к вершине оказывает тот же эффект, что и последовательное применение отдельных матриц.

Использовать матрицы не трудно. Для этого требуется лишь немного понимания. Фактически, благодаря мощи D3DX, вы можете легко комбинировать матрицы с помощью функции **D3DXMatrixMultiply**:

```
D3DXMATRIX *D3DXMatrixMultiply(
    D3DXMATRIX      *pOut, // Итоговая матрица
    CONST D3DXMATRIX *pM1, // Исходная матрица 1
    CONST D3DXMATRIX *pM2); // Исходная матрица 2
```

Вы передаете две матрицы в параметрах **pM1** и **pM2** и получаете итоговую матрицу **pOut**, вычисленную как результат умножения первых двух матриц. Остановимся на примере матриц перемещения, масштабирования и вращения, созданных в предыдущем разделе, и скомбинируем их вместе в одну матрицу, представляющую все преобразования.

```
D3DXMATRIX matResult; // Итоговая матрица

// Очищаем итоговую матрицу, делая ее единичной
D3DXMatrixIdentity(&matResult);

// Умножаем на матрицу масштабирования
D3DXMatrixMultiply(&matResult, &matResult, &matScale);

// Умножаем на матрицы вращения
D3DXMatrixMultiply(&matResult, &matResult, &matXRot);
D3DXMatrixMultiply(&matResult, &matResult, &matYRot);
D3DXMatrixMultiply(&matResult, &matResult, &matZRot);

// Умножаем на матрицу перемещения
D3DXMatrixMultiply(&matResult, &matResult, &matTrans);
```

Другой метод перемножения матриц между собой — использование оператора умножения (\*) объекта **D3DXMATRIX**, как показано ниже:

```
matResult = matXRot * matYRot * matZRot * matTrans;
```

**ПРИМЕЧАНИЕ**

Расширения `D3DXMATRIX`, такие как оператор умножения, позволяют работать с матрицами так, как будто они являются единым значением. Вы можете складывать, вычитать, умножать и даже делить матрицы используя показанный метод как будто матрица — это отдельное число.

Обратите внимание, что порядок комбинирования матриц очень важен. В предыдущем примере я комбинировал их в следующем порядке: масштабирование, поворот вокруг оси X, поворот вокруг оси Y, поворот вокруг оси Z и перемещение. Если вы будете комбинировать матрицы в любом другом порядке, полученная в результате матрица будет отличаться и может в дальнейшем привести к некоторым нежелательным результатам.

## Переход от локальных координат к координатам вида

Чтобы вершина могла использоваться для визуализации грани, она должна быть преобразована из локальных координат (непреобразованных координат) в мировые координаты. Мировые координаты надо преобразовать в координаты вида, а их в свою очередь спроецировать в двухмерные координаты (преобразованные координаты).

Вы преобразуете локальные координаты в мировые координаты с помощью *матрицы мирового преобразования* (*world transformation matrix*) или, для краткости, *мировой матрицы* (*world matrix*). Эта матрица содержит преобразования, применяемые для размещения объекта в трехмерном мире. Вторая матрица преобразования, которая используется для преобразования объекта в координаты вида, называется *матрицей вида* (*viewing matrix*). Последняя, *матрица проекции* (*projection matrix*), используется для преобразования трехмерных координат из координат вида в преобразованную вершину, которая применяется для визуализации графики.

Конструируя мировую матрицу и матрицу вида вы должны уделить самое пристальное внимание порядку в котором вы комбинируете отдельные матрицы. Для мирового преобразования вы комбинируете отдельные матрицы преобразований в следующем порядке:

$$R = S * X * Y * Z * T$$

**R** — это итоговая матрица, **S** — это матрица масштабирования, **X** — это матрица вращения вокруг оси X, **Y** — это матрица вращения относительно оси Y, **Z** — это матрица вращения относительно оси Z и **T** — это матрица перемещения. В матрице вида отдельные матрицы преобразования должны комбинироваться в следующем порядке (используя только перемещение и вращение):

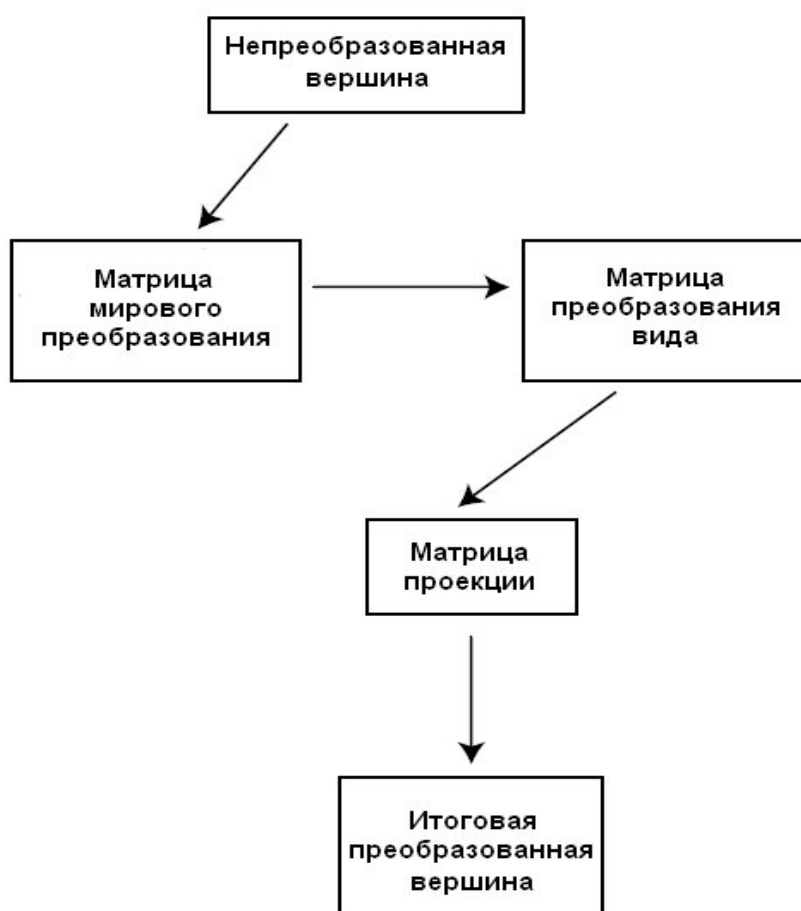
$$R = T * X * Y * Z$$

**ПРИМЕЧАНИЕ**

Работая с преобразованием вида вы должны использовать обратные значения местоположения точки просмотра для ориентации объектов в поле зрения. Это делается потому что точка просмотра в действительности зафиксирована в точке с координатами 0, 0, 0. Когда вы «перемещаете» точку просмотра, на самом деле все объекты мира перемещаются вокруг вас. Например, если вы хотите переместиться на 10 единиц вперед, вместо этого вам надо переместить все мировые объекты на 10 единиц к вам. Если вы хотите переместить взгляд на 10 градусов левее, надо повернуть все мировые объекты на 10 градусов вправо относительно вас.

Матрица проекции — это особый случай и работать с ней несколько труднее. Создавая матрицу проекции вы должны учесть множество вещей, поскольку она не имеет дела с такими преобразованиями, как перемещение, масштабирование или вращение. Я буду использовать библиотеку D3DX, которая помогает создавать матрицу проекции, о чем я расскажу позже в этой главе в разделе «Преобразование проекции».

На рис. 2.10 показан путь вершины через различные преобразования к заключительному набору координат для рисования.



*Рис. 2.10. Непреобразованная вершина проходит через различные матрицы преобразований для получения итоговых координат визуализации*



## Переходим к рисованию

Достаточно основ; пришло время посмотреть как Direct3D в действительности рисует графику. В этом разделе я расскажу об основах использования вершин и полигонов для рисования графики. Вы узнаете о различных способах, которыми Direct3D может использовать вершины для рисования полигонов, о том, как раскрасить эти полигоны, и, наконец, о том, как показать эту графику пользователю. Список достаточно короткий, так что давайте посмотрим, как работать с вершинами, и будем двигаться дальше.

### Использование вершин

Direct3D предоставляет вам свободу определять вершины различными способами. Например, если вы используете двухмерную графику, вы можете задавать координаты в двухмерной системе координат экрана (преобразованные координаты).

С другой стороны, если вы используете локальную или мировую систему координат, вы можете задавать координаты в трехмерном пространстве (непреобразованные координаты). А как обстоит дело с цветами и текстурами? Вы можете выбрать включать ли эту информацию в описание ваших вершин.

Как вы можете отслеживать всю эту информацию и гарантировать, что Direct3D знает, что вы делаете? Встречайте настраиваемый формат вершин.

### Настраиваемый формат вершин

*Настраиваемый формат вершин (flexible vertex format* или, для краткости, *FVF*) используется для конструирования произвольных структур данных вершины, которые будут применяться в вашем приложении. С FVF вы можете выбрать, какую информацию использовать для описания ваших вершин, — такую как трехмерные координаты, двухмерные координаты, цвет и т.д.

Вы конструируете FVF используя обычную структуру, в которую добавляете только те компоненты, которые вам требуются. Конечно, существуют некоторые ограничения, такие как необходимость перечислять компоненты в определенном порядке, и требование, чтобы отдельные компоненты не конфликтовали между собой (например, нельзя одновременно использовать трехмерные и двухмерные координаты). Как только структура создана, вы объявляете *descriptor FVF (FVF descriptor)*, являющийся комбинацией флагов, описывающих ваш формат вершин.

Приведенный ниже фрагмент кода содержит структуру данных вершины, использующую различные переменные, допустимые для FVF (по крайней мере те, которые я буду использовать в этой книге). Переменные в структуре перечислены именно в том порядке, в котором они должны располагаться в ваших собственных структурах; если вы убираете какую-

нибудь из переменных, убедитесь, что порядок остается тем, который показан:

```
typedef struct {
    FLOAT    x, y, z, rhw; // Двухмерные координаты
    FLOAT    x, y, z;      // Трехмерные координаты
    FLOAT    nx, ny, nz;    // Нормаль
    D3DCOLOR diffuse;      // Рассеиваемый цвет
    FLOAT    u, v;         // Координаты текстуры
} sVertex;
```

Как видите, единственные конфликтующие переменные — это координаты, в том числе и нормаль. *Нормаль (normal)* — это набор координат, определяющий направление, который может использоваться только совместно с трехмерными координатами. Вам необходимо выбрать, какой набор координат (двухмерные или трехмерные) оставить, а какой убрать. Если вы используете двухмерные координаты, то не можете включать трехмерные, и наоборот.

Единственным реальным различием между двухмерными и трехмерными координатами является добавление переменной **rhw**, которая является аналогом однородного W. Говоря человеческим языком, это значение обычно представляет расстояние от точки просмотра до вершины вдоль оси Z. В большинстве случаев вы спокойно можете присваивать переменной **rhw** значение 1.0.

Также обратите внимание, что структура **sVertex** использует данные типа **FLOAT** (числа с плавающей точкой), а что такое **D3DCOLOR**? **D3DCOLOR** — это значение типа **DWORD**, которое применяется в Direct3D для хранения значений цвета. Чтобы создать значение цвета типа **D3DCOLOR** можно воспользоваться одной из двух функций: **D3DCOLOR\_RGBA** или **D3DCOLOR\_COLORVALUE**:

```
D3DCOLOR D3DCOLOR_RGBA(Red, Green, Blue, Alpha);
D3DCOLOR D3DCOLOR_COLORVALUE(Red, Green, Blue, Alpha);
```

Каждая функция (в действительности это макросы) получает четыре параметра, каждый из которых задает величину отдельной цветовой составляющей, включая значение альфа-компоненты (прозрачность). Для макроса **D3DCOLOR\_RGBA** эти значения должны находиться в диапазоне от 0 до 255, а для макроса **D3DCOLOR\_COLORVALUE** — в диапазоне от 0.0 до 1.0 (быть десятичными дробями). Если вы используете сплошные цвета (непрозрачные), для альфа-компоненты всегда задавайте значение 255 (или 1.0).

В качестве примера предположим, что в вашу структуру данных вершины необходимо включить только трехмерные координаты и рассеиваемую составляющую цвета:

```
typedef struct {
    FLOAT    x, y, z;
    D3DCOLOR diffuse;
} sVertex;
```

Следующим этапом конструирования вашего FVF является создание дескриптора FVF с использованием комбинации флагов, перечисленных в таблице 2.3.

**Таблица 2.3.** Флаги дескриптора настраиваемого формата вершин

Флаг	Описание
D3DFVF_XYZ	Включены трехмерные координаты
D3DFVF_XYZRHW	Включены двухмерные координаты
D3DFVF_NORMAL	Включена нормаль (вектор)
D3DFVF_DIFFUSE	Включена рассеиваемая составляющая цвета
D3DFVF_TEX1	Включен один набор координат текстуры

Чтобы описать дескриптор FVF вы комбинируете соответствующие флаги в определении (подразумевается, что вы используете трехмерные координаты и рассеиваемую составляющую цвета):

```
#define VertexFVF (D3DFVF_XYZ | D3DFVF_DIFFUSE)
```

Просто убедитесь, что все флаги соответствуют компонентам, которые вы добавили в вашу структуру данных вершины, и все пойдет гладко.

## Использование буфера вершин

После того, как вы создали вашу структуру данных вершины и дескриптор, вы создаете объект, который содержит массив вершин. Direct3D предоставляет вам для работы два объекта: **IDirect3DVertexBuffer9** и **IDirect3DIndexBuffer9**. В этой книге я буду использовать объект **IDirect3DVertexBuffer9**, который хранит вершины, используемые для рисования списков треугольников, полос треугольников и вееров треугольников.

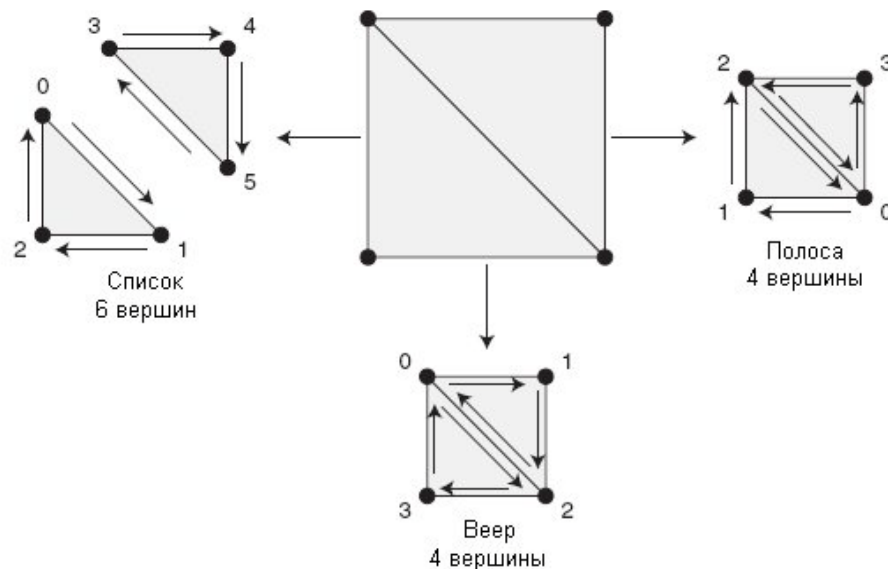
При работе со списками треугольников объект **IDirect3DVertexBuffer9** хранит как минимум по три вершины для каждого рисуемого полигона (вершины упорядочены по часовой стрелке). У полосы треугольников для рисования первого полигона используется три вершины, а для рисования каждого последующего полигона используется одна дополнительная вершина. Для веера треугольников задается одна центральная вершина, а для рисования каждого полигона хранятся две дополнительные вершины.

### ПРИМЕЧАНИЕ

Полигон может использовать одну, две или три вершины, в зависимости от того, что именно вы рисуете. Для точек требуется одна единственная вершина, для линий необходимо две, а для треугольников — три вершины. В этой книге мы в основном будем иметь дело с треугольными полигонами.

Рис. 2.11 поможет вам лучше понять, как хранятся вершины и как их надо упорядочивать. На рисунке показан квадрат, представленный тремя разными способами. Первый способ — использование списка треугольников, который требует для представления квадрата шесть вершин — по три вершины для каждого из двух треугольников.

Второй способ представления квадрата — использование полосы треугольников. Полоса треугольников использует только четыре вершины, упорядоченные, как показано на рисунке. Первые три вершины образуют первую грань, а последняя вершина используется для формирования второй грани. Для третьего метода упорядочивания, веера треугольников, также используется четыре вершины. Однако, в случае с веером, первая вершина задает основание веера, а последующие вершины образуют грани.



**Рис. 2.11.** Вы можете хранить простой полигон, такой как квадрат (с четырьмя вершинами и двумя полигонами), различными способами. В зависимости от того, как хранятся вершины, для описания квадрата требуется до шести вершин

## Создание буфера вершин

Вы создаете буфер вершин, используя инициализированный объект **IDirect3DDevice9**:

```
HRESULT IDirect3DDevice9::CreateVertexBuffer(
    UINT Length,           // Количество вершин, умноженное на
                           // размер структуры данных вершины
    DWORD Usage,           // 0
    DWORD FVF,             // Deskriptor FVF
    D3DPool Pool,          // D3DPool_MANAGED
    IDirect3DVertexBuffer **ppVertexBuffer, // Буфер вершин
    HANDLE *pHandle);     // укажите NULL
```

В вызове **CreateVertexBuffer** есть только один параметр, значение которого вы, возможно, решите изменить — это флаг **Usage**, который сообщает Direct3D как будет использоваться память, предназначенная для хранения вершин. Вполне безопасно всегда

присваивать **Usage** значение 0, но, если вы хотите несколько увеличить производительность, присвойте **Usage** значение **D3DCREATE\_WRITEONLY**. Сделав так вы проинформируете Direct3D, что не планируете читать данные вершин из буфера и все должно быть хорошо, если в соответствии с этим выбрать место хранения. Обычно это означает, что данные вершин будут размещены в памяти видеокарты (читай: памяти с более быстрым доступом).

Вот небольшой пример создания буфера вершин, содержащего четыре вершины (построенный на основании созданного ранее в разделе «Настраиваемый формат вершин» формата вершин, который содержит только трехмерные координаты и рассеиваемую составляющую цвета):

```
// g_pD3DDevice = ранее инициализированный объект устройства
// sVertex       = ранее определенная структура данных вершины
// VertexFVF     = ранее определенный дескриптор FVF
IDirect3DVertexBuffer9 *pD3DVB = NULL;

// Создание буфера вершин
if(FAILED(g_pD3DDevice->CreateVertexBuffer(sizeof(sVertex) * 4,
                                           D3DCREATE_WRITEONLY, VertexFVF,
                                           D3DPOOL_MANAGED, &pD3DVB, NULL))) {
    // Произошла ошибка
}
```

---

**ПРИМЕЧАНИЕ** Когда вы создаете собственный буфер вершин, убедитесь, что указали требуемый размер буфера (первый параметр в вызове **CreateVertexBuffer**). В показанном выше примере выделяется память для хранения четырех вершин типа **sVertex**. 4 означает четыре экземпляра, а **sVertex** — это структура, которую вы используете для хранения данных ваших вершин.

---

---

**ПРИМЕЧАНИЕ** Как обычно, убедитесь, что по завершении работы вы освобождаете COM-объект буфера вершин, вызвав его метод **Release**.

---

### **Блокировка буфера вершин**

Перед добавлением вершин к объекту буфера вершин, вы должны заблокировать память, которую использует буфер. Это гарантирует, что память, используемая для хранения данных вершин, будет доступна. Затем вы используете указатель на область памяти для доступа к памяти буфера вершин. Вы блокируете память буфера вершин и получаете указатель на нее с помощью вызова метода **Lock** объекта буфера:

```
HRESULT IDirect3DVertexBuffer9::Lock(
    UINT OffsetToLock, // Смещение до начала блокируемой области,
                        // в байтах
    UINT SizeToLock,   // Сколько байт блокировать, 0 = все
    VOID **ppbData,    // Указатель на указатель (для доступа к данным)
    DWORD Flags);      // 0
```

Вы указываете смещение от начала буфера до той области к которой хотите получить доступ (в байтах), а также размер блокируемой области в байтах (0, если блокируете весь буфер). Затем вам надо предоставить функции указатель на указатель, который вы будете использовать для доступа к буферу вершин (приведенный к типу **VOID**). Вот пример вызова, который блокирует весь буфер вершин:

```
// pD3DVB = ранее инициализированный объект буфера вершин
BYTE *Ptr;

// Блокировка памяти буфера вершин и получение указателя на нее
if(FAILED(pD3DVB->Lock(0, 0, (void**)&Ptr, 0))) {
    // Произошла ошибка
}
```

### ВНИМАНИЕ!

Буфер вершин при создании которого был указан флаг **D3DCREATE\_WRITEONLY** недоступен для чтения, а доступен только для записи. Если вы планируете читать данные из буфера вершин (как это делают некоторые функции библиотеки D3DX), то в вызове **CreateVertexBuffer** должны присвоить параметру **Usage** значение 0.

После того, как вы завершили доступ к буферу вершин для каждого вызова **Lock** вызывайте **IDirect3DVertexBuffer9::Unlock**:

```
HRESULT IDirect3DVertexBuffer9::Unlock();
```

Разблокировка буфера вершин гарантирует, что Direct3D может безопасно начать его использование, зная, что вы не будете модифицировать никакие содержащиеся в буфере данные.

### ВНИМАНИЕ!

Всегда минимизируйте время, проходящее между вызовами **Lock** и **Unlock**. Чем быстрее вы завершите работу с заблокированным буфером вершин, тем лучше, поскольку Direct3D ждет, пока вы не закончите свои дела, чтобы получить доступ к содержащимся в буфере данным.

## Заполнение данных вершин

Теперь у вас есть структура данных вершины, дескриптор и буфер, который вы заблокировали, подготовив тем самым к сохранению данных вершин. Поскольку с помощью вызова **Lock** вы уже получили указатель на память буфера вершин, все что вам необходимо — это скопировать требуемое количество вершин в буфер вершин.

Продолжая мой пример и используя определенный ранее формат вершин (который содержит трехмерные координаты и рассеиваемую составляющую цвета), я создаю локальный набор данных вершин в массиве:

```
sVertex Verts[4] = {
    { -100.0f, 100.0f, 100.0f, D3DCOLOR_RGBA(255,255,255,255) },
    { 100.0f, 100.0f, 100.0f, D3DCOLOR_RGBA(255, 0, 0,255) },
    { 100.0f, -100.0f, 100.0f, D3DCOLOR_RGBA( 0,255, 0,255) },
    { -100.0f, -100.0f, 100.0f, D3DCOLOR_RGBA( 0, 0,255,255) }
};
```

Заблокируйте буфер вершин, получив тем самым указатель на память буфера вершин, и скопируйте локальные данные вершин (и по завершении разблокируйте буфер):

```
// pD3DVB = ранее инициализированный объект буфера вершин
BYTE *Ptr;

// Блокируем память буфера вершин и получаем указатель на нее
if(SUCCEEDED(pD3DVB->Lock(0, 0, (void**)&Ptr, 0))) {

    // Копируем локальные вершины в буфер вершин
    memcpy(Ptr, Verts, sizeof(Verts));

    // Разблокируем буфер вершин
    pD3DVB->Unlock();
}
```

Вот и все, что касается создания буфера вершин и заполнения его данными! Теперь вам осталось только назначить источник потоковых данных и задать вершинный шейдер для обработки информации о вершинах.

## Поток вершин

Direct3D позволяет вам передавать вершины визуализатору через несколько каналов, называемых *потоки вершин* (*vertex stream*). Объединяя несколько потоков вершин в единый поток вы можете добиться потрясающих результатов, но в данной книге я использую только один поток, поскольку сложности использования нескольких потоков выходят за рамки этой книги.

Чтобы указать, какие данные вершин будут передаваться в поток, вы используете функцию **IDirect3DDevice9::SetStreamSource**:

```
HRESULT IDirect3DDevice9::SetStreamSource(
    UINT StreamNumber, // 0
    IDirect3DVertexBuffer9* pStreamData, // Объект буфера вершин
    UINT OffsetInBytes, // Смещение (в байтах) данных вершин
                        // в буфере вершин
    UINT Stride);       // Размер структуры данных вершины
```

Все, что необходимо делать для установки источника потока вершин — вызвать данную функцию, передав ей указатель на объект буфера вершин и указав количество байт, используемых для хранения структуры данных вершины (с помощью **sizeof**).

Если в буфере вершин вы храните более одной группы полигонов (например, несколько полос треугольников), вы можете в параметре **OffsetInBytes** указать смещение в байтах до начала требуемой группы вершин. Например, если я использую буфер вершин для хранения двух полос треугольников (первая начинается со смещения 0, а вторая — со смещения 6), я могу нарисовать вторую полосу треугольников, указав соответствующее смещение (в рассматриваемом примере — 6).

Для моего примера работы с буфером вершин, приведенного в предыдущем разделе, вы должны использовать следующий код:

```
// g_pD3DDevice = ранее инициализированный объект устройства
// pD3DVB       = ранее инициализированный буфер вершин
if(FAILED(g_pD3DDevice->SetStreamSource(0, pD3DVB,
                                         0, sizeof(sVertex)))) {
    // Произошла ошибка
}
```

## Вершинные шейдеры

Для выполнения заключительного этапа в использовании вершин для рисования графики вам необходимо познакомиться с концепцией вершинных шейдеров. *Вершинный шейдер (vertex shader)* — это механизм, выполняющий загрузку и обработку вершин; в это входит модификация координат вершины, применение цветов и тумана, и обработка прочих компонентов данных вершины.

Существует две формы вершинных шейдеров. Это может быть *фиксированный (fixed)* вершинный шейдер (в который встроена вся функциональность, необходимая в подавляющем большинстве случаев) или *программируемый (programmable)* вершинный шейдер (в котором вы пишете собственную процедуру для модификации данных вершины перед визуализацией на экране).

Исследование программируемых вершинных шейдеров выходит за рамки этой книги. Вместо этого я сосредоточусь на применении фиксированных вершинных шейдеров, поскольку они содержат всю функциональность, которая может нам понадобиться.

Чтобы использовать фиксированный вершинный шейдер для ваших вершин, вам надо передать свой дескриптор FVF вершины в функцию **IDirect3DDevice9::SetFVF**:

```
HRESULT IDirect3DDevice9::SetFVF(
    DWORD FVF); // Дескриптор FVF вершины
```

Использовать показанную функцию очень просто:

```
// g_pD3DDevice = ранее инициализированный объект устройства
// VertexFVF     = ранее определенный дескриптор FVF вершины
if(FAILED(g_pD3DDevice->SetFVF(VertexFVF))) {
    // Произошла ошибка
}
```

Вы указали данные о вершинах. Теперь надо указать различные преобразования, необходимые для размещения вершины (определенной в ее локальном пространстве) в мировой системе координат. Конечно, это необходимо делать только в том случае, если вы используете трехмерные координаты.

## Преобразования

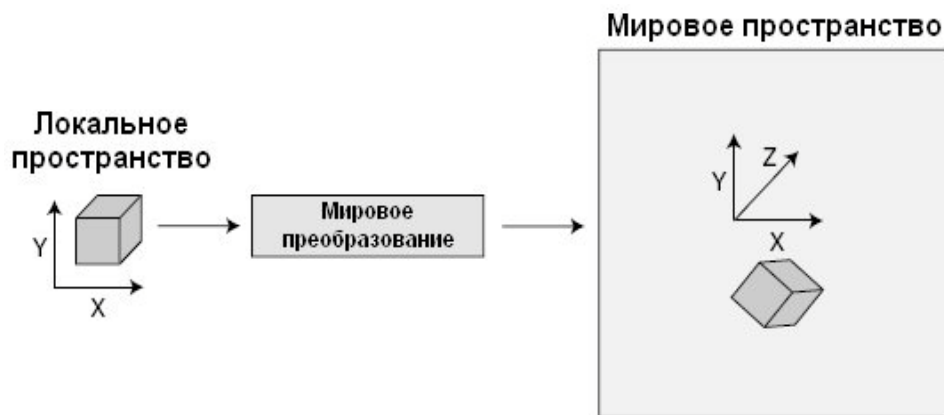
К данному моменту вы узнали как инициализировать графическую систему и создавать вершины. Если вы имеете дело с трехмерными объектами, такими



как полигоны, вершины, вероятно, будут определены в локальном пространстве. Если так, вы проводите вершины через несколько преобразований (мировое, вида и проекции), чтобы гарантировать, что к моменту визуализации объекта они будут правильно размещены. Каждое преобразование требует конструирования специальной матрицы, которая содержит значения, представляющие требуемую ориентацию (или проекцию). В нескольких следующих разделах я покажу вам, как конструировать и использовать каждое из этих трех преобразований, и начнем мы с мирового преобразования.

## Мировое преобразование

Вершинам, которые определены в локальном пространстве, требуется преобразование в соответствующие координаты мирового пространства. Например, если вы создаете из вершин куб (в локальном пространстве) и хотите поместить его в выбранное вами место в мире, то вы применяете к нему мировое преобразование (как показано на рис. 2.12).



*Рис. 2.12. Куб, созданный в локальном пространстве, необходимо перед визуализацией ориентировать в мировом пространстве*

Используйте старого знакомого, библиотеку D3DX, которая поможет вам создать матрицу мирового преобразования. Для ориентации объекта вам потребуется сконструировать три матрицы вращения (по одной для каждой оси), матрицу перемещения и матрицу масштабирования.

```
D3DXMATRIX matWorld;
D3DXMATRIX matRotX, matRotY, matRotZ;
D3DXMATRIX matTrans;
D3DXMATRIX matScale;

// Создаем матрицы вращения
D3DXMatrixRotationX(&matRotX, XAngle);
D3DXMatrixRotationY(&matRotY, YAngle);
D3DXMatrixRotationZ(&matRotZ, ZAngle);

// Создаем матрицу перемещения
D3DXMatrixTranslation(&matTrans, XPos, YPos, ZPos);

// Создаем матрицу масштабирования
D3DXMatrixScaling(&matScale, XScale, YScale, ZScale);
```

Затем вы комбинируете все эти матрицы в матрицу мирового преобразования. Комбинировать матрицы необходимо в следующем порядке: масштабирование, поворот относительно оси X, поворот относительно оси Y, поворот относительно оси Z, и затем перемещение.

```
// Делаем матрицу matWorld единичной
D3DXMatrixIdentity(&matWorld);

// Комбинируем все матрицы в матрицу мирового преобразования
D3DXMatrixMultiply(&matWorld, &matWorld, &matScale);
D3DXMatrixMultiply(&matWorld, &matWorld, &matRotX);
D3DXMatrixMultiply(&matWorld, &matWorld, &matRotY);
D3DXMatrixMultiply(&matWorld, &matWorld, &matRotZ);
D3DXMatrixMultiply(&matWorld, &matWorld, &matTrans);
```

Почти все готово. Теперь осталось только сообщить Direct3D, что ему надо использовать матрицу мирового преобразования, которую вы только что создали. Делается это с помощью следующей функции:

```
HRESULT IDirect3DDevice9::SetTransform(
    D3DTRANSFORMSTATETYPE State,      // D3DTS_WORLD
    CONST D3DMATRIX *pMatrix); // Устанавливаемая мировая матрица
```

Обратите внимание, что второй параметр — это указатель на структуру **D3DMATRIX**, но, к счастью, вы можете здесь использовать объект **D3DXMATRIX**, который создали. Значение **D3DTS\_WORLD** в первом параметре сообщает Direct3D, что матрица используется для мирового преобразования и все, что будет рисоваться в дальнейшем, должно быть ориентировано с использованием этой матрицы.

Если вам необходимо расположить в мире несколько объектов, просто создайте матрицу мирового преобразования для размещения каждого из них (в требуемой ориентации) и затем снова вызовите **SetTransform**, убедившись перед переходом к следующему мировому преобразованию, что все требуемые объекты нарисованы.

## Преобразование вида

Фактически преобразование вида действует как камера (называемая *точка просмотра* — *viewpoint*). Создавая матрицу, которая содержит смещения для ориентации вершин в мировом пространстве, вы выравниваете сцену относительно точки просмотра. Все вершины должны быть ориентированы относительно центра мира (с использованием преобразования вида) точно в те же относительные позиции, в которых они находятся относительно точки просмотра.

Чтобы создать преобразование вида, вы строите матрицу из перемещения и поворота точки просмотра, располагая преобразования в следующем порядке: перемещение, поворот вокруг оси Z, поворот вокруг оси Y и поворот вокруг оси X. Трюк здесь заключается в том, что вы используете для местоположения и поворота противоположные значения.

Например, если точка просмотра расположена в координатах  $X = 10$ ,  $Y = 0$ ,  $Z = -150$ , вы используете значения  $X = -10$ ,  $Y = 0$ ,  $Z = 150$ .

Вот как выглядит код построения матрицы преобразования вида:

```
D3DXMATRIX matView;
D3DXMATRIX matRotX, matRotY, matRotZ;
D3DXMATRIX matTrans;

// Создаем матрицы вращения (с противоположными значениями)
D3DXMatrixRotationX(&matRotX, -XAngle);
D3DXMatrixRotationY(&matRotY, -YAngle);
D3DXMatrixRotationZ(&matRotZ, -ZAngle);

// Создаем матрицу перемещения (с противоположными значениями)
D3DXMatrixTranslation(&matTrans, -XPos, -YPos, -ZPos);

// Инициализируем единичную матрицу matView
D3DXMatrixIdentity(&matView);

// Комбинируем все матрицы в матрицу преобразования вида
D3DXMatrixMultiply(&matView, &matView, &matTrans);
D3DXMatrixMultiply(&matView, &matView, &matRotZ);
D3DXMatrixMultiply(&matView, &matView, &matRotY);
D3DXMatrixMultiply(&matView, &matView, &matRotX);
```

Чтобы Direct3D использовал созданную вами матрицу преобразования вида, снова воспользуйтесь функцией **IDirect3DDevice9::SetTransform**, но на этот раз в параметре **State** укажите значение **D3DTS\_VIEW**:

```
// g_pD3DDevice = ранее инициализированный объект устройства
if (FAILED(g_pD3DDevice->SetTransform(D3DTS_VIEW, &matView))) {
    // Произошла ошибка
}
```

Как видите, установить преобразование вида просто; основную проблему представляет конструирование матрицы вида. Чтобы сделать жизнь проще, D3DX предоставляет функцию, которая за один вызов инициализирует матрицу преобразования вида:

```
D3DXMATRIX* D3DXMatrixLookAtLH(
    D3DXMATRIX *pOut, // Итоговая матрица преобразования вида
    CONST D3DXVECTOR *pEye, // Координаты точки просмотра
    CONST D3DXVECTOR *pAt, // Координаты цели
    CONST D3DXVECTOR *pUp); // Верхнее направление
```

Разобраться с первого взгляда в функции **D3DXMatrixLookAtLH** не так-то просто. Вы видите обычный указатель для возврата вычисленной матрицы, но что это за три объекта **D3DXVECTOR3**? **D3DXVECTOR3** во многом похож на объект **D3DXMATRIX**, за исключением того, что содержит лишь три значения (называемые **x**, **y** и **z**) — в данном случае три значения координат. Такой объект **D3DXVECTOR3** называется *объектом вектора* (*vector object*).

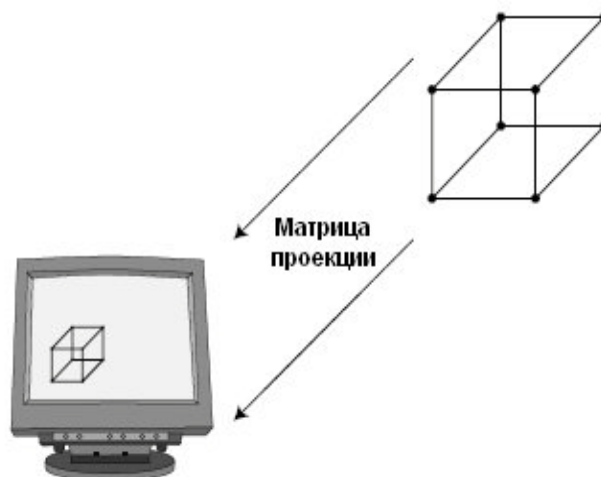
Параметр **pEye** представляет координаты точки просмотра, а **pAt** представляет координаты цели, то есть той точки, на которую направлен взгляд. Параметр **pUp** — это вектор, задающий направление вверх для точки просмотра. Обычно для **pUp** задаются значения 0, 1, 0 (это означает, что направление вверх совпадает с положительным направлением оси Y), но поскольку точка просмотра может наклоняться (точно так же, как вы наклоняете голову из стороны в сторону), направление вверх может указывать в любом направлении вдоль любой из осей.

Чтобы применить функцию **D3DXMatrixLookAtLH** вы можете воспользоваться следующим фрагментом кода (подразумевается, что точка просмотра имеет координаты **XPos**, **YPos**, **ZPos**, а взгляд направлен на начало координат):

```
D3DXMATRIX matView;
D3DXVECTOR3 vecVP, vecTP, vecUp(0.0f, 1.0f, 0.0f);
vecVP.x = XPos;
vecVP.y = YPos;
vecVP.z = ZPos;
vecTP.x = vecTP.y = vecTP.z = 0.0f;
D3DXMatrixLookAtLH(&matView, &vecVP, &vecTP, &vecUp);
```

## Преобразование проекции

Последним выполняется преобразование проекции, преобразующее трехмерные вершины (непреобразованные) в двухмерные координаты (преобразованные), которые Direct3D использует для рисования вашей графики на экране. Думайте о преобразовании проекции, как о способе расплющивания трехмерной графики на экране вашего дисплея (как показано на рис. 2.13).



*Рис. 2.13. Преобразование проекции позволяет видеть объекты, определенные с использованием трехмерных координат, на плоском двухмерном экране*

Имея дело с преобразованием проекции следует учесть множество аспектов, таких как соотношение размеров порта просмотра, поле зрения, а также ближнюю и дальнюю плоскости отсечения.

Что за отсечение? При рисовании трехмерной графики некоторые объекты могут оказаться расположены слишком близко к точке просмотра или слишком далеко от нее. Вы указываете Direct3D, когда можно отбросить эти объекты (для увеличения скорости). Чтобы сконструировать матрицу проекции и определить область пространства, в которой объекты видимы и не могут быть отсечены, мы будем использовать функцию **D3DXMatrixPerspectiveFovLH**:

```
D3DXMATRIX* D3DXMatrixPerspectiveFovLH(
    D3DXMATRIX *pOut, // Итоговая матрица
    FLOAT fovy, // Угол поля зрения в радианах
    FLOAT Aspect, // Соотношение размеров
    FLOAT zn, // Координата Z ближней плоскости отсечения
    FLOAT zf); // Координата Z дальней плоскости отсечения
```

---

**ПРИМЕЧАНИЕ** Функция **D3DXMatrixPerspectiveFovLH** создает матрицу перспективной проекции для левосторонней системы координат. Если вы используете правостороннюю систему координат, воспользуйтесь функцией **D3DXMatrixPerspectiveFovRH** (у нее те же параметры, что и у версии для левосторонней системы).

---

Параметр **fovy** указывает ширину проектируемого вида, поэтому чем большее число вы укажете, тем больше будете видеть. Но это обоюдоострый меч, поскольку если вы используете слишком маленькие или слишком большие значения, представление становится искаженным. Обычным значением для **fovy** является **D3DX\_PI/4**, то есть одна четвертая пи.

Следующий важный параметр — **Aspect**, который является соотношением размеров области просмотра. Если у вас есть окно размером 400 × 400 пикселей, соотношение размеров будет 1 : 1 или 1.0 (потому что это квадрат). Если у вас окно размером 400 × 200 точек (высота в два раза меньше ширины) соотношение размеров будет 2 : 1 или 2.0. Чтобы вычислить это значение разделите ширину окна на его высоту:

```
FLOAT Aspect = (FLOAT)WindowWidth / (FLOAT)WindowHeight;
```

Параметры **zn** и **zf** — это координаты местоположения ближней и дальней плоскостей отсечения, измеренные в тех же единицах, которые используются для описания местоположения вершин в трехмерном пространстве. Обычно для ближней и дальней плоскостей отсечения задаются значения 1.0 и 1000.0 соответственно. Эти два значения означают, что полигоны, расположенные менее чем на 1.0 единицу (или далее, чем на 1000.0 единиц) относительно точки просмотра не будут рисоваться. Вы можете указать для параметра **zf** большее значение, если в вашем проекте необходимо рисовать объекты, удаленные более чем на 1000 единиц от камеры.

После создания матрицы проекции вы устанавливаете ее с помощью функции **IDirect3DDevice9::SetTransform**, указывая в параметре **State** значение **D3DTS\_PROJECTION**:

```
// g_pD3DDevice = ранее инициализированный объект устройства
D3DXMATRIX matProj;

// Создаем матрицу преобразования проекции
D3DXMatrixPerspectiveFovLH(&matProj, D3DX_PI/4, 1.0f, 1.0f, 1000.0f);

// Устанавливаем матрицу проекции для Direct3D
if(FAILED(g_pD3DDevice->SetTransform(D3DTS_PROJECTION, &matProj))) {
    // Произошла ошибка
}
```

## Материалы и цвета

Вы уже видели, как можно задать цвета в данных вершин, но на уровне полигонов можно также задавать специальные параметры цвета. Цвета, применяемые для граней объекта, называются *материалами* (*materials*). Перед тем, как рисовать полигон с помощью Direct3D, вы можете назначить материал, который будет использоваться (если вы решите не использовать материалы, можно использовать цвета вершин, если они заданы).

У каждого материала есть набор описывающих его цветовых компонентов. В Direct3D описывающие материал цветовые составляющие хранятся в структуре:

```
typedef struct _D3DMATERIAL9 {
    D3DCOLORVALUE Diffuse; // Рассеиваемая составляющая цвета
    D3DCOLORVALUE Ambient; // Фоновая составляющая цвета
    D3DCOLORVALUE Specular; // Отражаемая составляющая цвета
    D3DCOLORVALUE Emissive; // Испускаемая составляющая цвета
    float         Power;    // Контрастность бликов
} D3DMATERIAL9;
```

В реальной жизни вы в большинстве случаев будете иметь дело только с одной составляющей: **Diffuse**. Значение составляющей **Ambient** обычно то же самое, что у **Diffuse**, а для **Specular** вы можете задать значения 0.0 или 1.0 (установив значение **Power** равным 0.0). Я предлагаю вам немного поэкспериментировать со значениями, чтобы понять какой эффект производит каждый из компонентов.

Для текущих целей вам достаточно использовать для грани только компонент **Diffuse**; цвет материала используется вместо рассеиваемой составляющей цвета вершины. Если для грани вы используете цвет материала и одновременно задаете цвета вершин, то можете получить ощутимые (и зачастую нежелательные) изменения цвета полигона. Поэтому лучше всего использовать либо материал, либо цвет вершин, но не оба вместе.

Имея дело с цветовыми составляющими материала, вы задаете значение каждого компонента непосредственно, а не используете макросы наподобие **D3DCOLOR\_RGBA**. Не беспокойтесь — каждая компонента цвета представляется первой буквой ее английского названия (**r** для красного, **g** для зеленого, **b** для синего и **a** для альфа) и диапазон значений тот же (от 0.0

до 1.0). Если вы хотите создать материал желтого цвета, то инициализация структуры данных материала должна выглядеть так:

```
D3DMATERIAL9 d3dm;

// Очищаем структуру данных материала
ZeroMemory(&d3dm, sizeof(D3DMATERIAL9));

// Задаем для компонентов Diffuse и Ambient желтый цвет
d3dm.Diffuse.r = d3dm.Ambient.r = 1.0f; // Красный
d3dm.Diffuse.g = d3dm.Ambient.g = 1.0f; // Зеленый
d3dm.Diffuse.b = d3dm.Ambient.b = 0.0f; // Синий
d3dm.Diffuse.a = d3dm.Ambient.a = 1.0f; // Альфа
```

Инициализировать структуру материала вы можете так, как хотите, но, после того как структура инициализирована, необходимо до визуализации полигонов сказать Direct3D, чтобы он использовал ее. Эту задачу выполняет функция **IDirect3DDevice9::SetMaterial**, которая в единственном параметре получает указатель на структуру данных вашего материала:

```
IDirect3DDevice9::SetMaterial(CONST D3DMATERIAL9 *pMaterial);
```

После этого вызова все визуализируемые полигоны будут отображаться с учетом установленных параметров материала. Вот пример установки ранее созданного материала желтого цвета:

```
g_pD3DDevice->SetMaterial(&d3dm);
```

## Очистка порта просмотра

Вам необходимо очистить вторичный буфер для подготовки его к рисованию, стерев графику, которая может быть в нем. Эта задача выполняется функцией **IDirect3DDevice::Clear**:

```
HRESULT IDirect3DDevice9::Clear(
    DWORD          Count,      // 0
    CONST D3DRECT *pRects,    // NULL
    DWORD          Flags,      // D3DCLEAR_TARGET
    D3DCOLOR       Color,      // Цвет для очистки
    float          Z,          // 1.0f
    DWORD          Stencil);   // 0
```

Единственный параметр, которому надо уделить внимание сейчас, это **Color**, определяющий цвет, которым будет заполнен вторичный буфер. Значение цвета может формироваться с помощью обычных макросов **D3DCOLOR\_RGBA** или **D3DCOLOR\_COLORVALUE**, с которыми вы уже познакомились. Предположим, вы хотите очистить вторичный буфер, заполнив его светло-синим цветом:

```
// g_pD3DDevice = ранее инициализированный объект устройства
if(FAILED(g_pD3DDevice->Clear(0, NULL, D3DCLEAR_TARGET,
    D3DCOLOR_RGBA(0,0,192,255), 1.0f, 0))) {
    // Произошла ошибка
}
```

---

**ПРИМЕЧАНИЕ** Флаг **D3DCLEAR\_TARGET** используется чтобы сообщить Direct3D, что именно вы хотите очистить (экран или окно). Есть несколько доступных для использования полезных флагов и с некоторыми из них вы вскоре познакомитесь.

---

## Начало и завершение сцены

Перед тем, как вы будете что-либо визуализировать, необходимо сообщить Direct3D о том, что надо подготовиться к визуализации. Для этой цели используется функция **IDirect3DDevice9::BeginScene** (у которой нет параметров):

```
HRESULT IDirect3DDevice9::BeginScene();
```

Когда вы завершаете визуализацию сцены, вам необходимо уведомить об этом Direct3D, воспользовавшись функцией **EndScene**:

```
HRESULT IDirect3DDevice9::EndScene();
```

Вы не должны помещать вызов функции **Clear** между **BeginScene** и **EndScene**; эта функция должна вызываться до **BeginScene**. Между вызовами функций начала и завершения сцены можно помещать только вызовы функций, рисующих полигоны.

## Визуализация полигонов

Наконец-то вы готовы к визуализации полигонов! Обычный кадр в вашем движке игры состоит из очистки вторичного буфера, начала сцены, установки используемых материалов, рисования полигонов и завершения сцены. Вы уже видели выполнение всех этих действий за исключением рисования полигонов.

Вы рисуете объект **IDirect3DVertexBuffer9** с помощью следующей функции (конечно, после того как вы установили источник данных вершин и задали их формат):

```
HRESULT IDirect3DDevice9::DrawPrimitive(
    D3DPRIMITIVETYPE PrimitiveType,    // Рисуемые примитивы
    UINT StartVertex,                  // Начальная вершина (0)
    UINT PrimitiveCount);              // Количество рисуемых примитивов
```

Первый параметр, **PrimitiveType**, сообщает Direct3D какие именно фигуры надо рисовать (список некоторых из них приведен в таблице 2.4). Параметр **StartVertex** позволяет указать, с какой именно вершины следует начать рисование (обычно указывается 0). В параметре **PrimitiveCount** вы указываете сколько именно примитивов (полигонов) следует нарисовать.

Используемый тип примитива зависит от того, как вы заполняли буфер данными вершин. Если вы использовали по три вершины для каждого полигона, укажите тип **D3DPT\_TRIANGLELIST**. Если вы применяете более



эффективный способ, например, полосу треугольников, укажите тип **D3DPT\_TRIANGLESTRIP**.

**Таблица 2.4.** Типы примитивов в параметре DrawPrimitive

<i>Тип</i>	<i>Описание</i>
<b>D3DPT_POINTLIST</b>	Рисуем все вершины как пикселы
<b>D3DPT_LINELIST</b>	Рисуем набор изолированных линий, каждая из которых использует две вершины
<b>D3DPT_LINESTRIP</b>	Рисуем набор линий, каждая из которых (кроме первой) связана с предыдущей
<b>D3DPT_TRIANGLELIST</b>	Рисуем отдельные полигоны, используя три вершины для каждого из них
<b>D3DPT_TRIANGLESTRIP</b>	Рисуем полосу треугольников, используя для первого из них три вершины, а для каждого последующего — две вершины из предыдущего треугольника и одну дополнительную
<b>D3DPT_TRIANGLEFAN</b>	Рисуем веер из полигонов, используя первую вершину в качестве центра (все остальные полигоны присоединены к ней)

Следует помнить только об одной вещи — перед тем, как визуализировать полигоны, вы должны начать сцену с помощью функции **IDirect3DDevice9::BeginScene**; иначе вызов функции **DrawPrimitive** приведет к ошибке.

Предположим, вы создали буфер вершин, содержащий шесть вершин, образующих два треугольных полигона, которые формируют квадрат. Их визуализация (вместе с вызовами функций **BeginScene** и **EndScene**, а также установкой источника данных вершин и формата данных вершин) будет выглядеть примерно так:

```
// g_pD3DDevice = ранее инициализированный объект устройства
// pD3DVB       = ранее инициализированный буфер вершин
// sVertex       = ранее созданная структура данных вершины
// VertexFVF     = ранее объявленный дескриптор FVF

// Устанавливаем источник потока данных и шейдер
g_pD3DDevice->SetStreamSource(0, pD3DVB, 0, sizeof(sVertex));
g_pD3DDevice->SetFVF(VertexFVF);

if (SUCCEEDED(g_pD3DDevice->BeginScene())) {

    // Визуализируем полигоны
    if (FAILED(g_pD3DDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 2))) {
        // Произошла ошибка
    }

    // Завершаем сцену
    g_pD3DDevice->EndScene();
}
```

## Показ сцены

Наконец-то вы готовы переключить вторичный буфер и экранную поверхность, чтобы показать пользователю созданную вами графику с помощью следующей функции:

```
HRESULT IDirect3DDevice9::Present(
    CONST RECT      *pSourceRect,
    CONST RECT      *pDestRect,
    HWND            hDestWindowOverride,
    CONST RGNDATA   *pDirtyRegion);
```

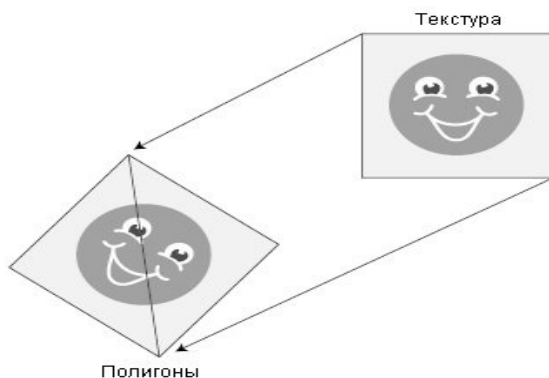
Абсолютно спокойно можете указывать для всех параметров функции **IDirect3DDevice9::Present** значения **NULL**, что сообщит Direct3D о необходимости обновить весь экран (поскольку функция может обновлять и отдельные фрагменты экрана), как показано в следующем фрагменте кода:

```
// g_pD3DDevice = ранее инициализированный объект устройства
if (FAILED(g_pD3DDevice->Present(NULL, NULL, NULL, NULL))) {
    // Произошла ошибка
}
```

Вот и все! Чтобы создавать более реалистичные сцены вы используете несколько различных буферов вершин для рисования различных трехмерных объектов в вашем мире. Другим способом увеличения реализма вашей графики является использование наложения текстур.

## Использование наложения текстур

Хотя вы и научились рисовать трехмерные объекты на экране, одноцветные полигоны достаточно скучны. Пришло время сделать вещи интереснее и добавить несколько деталей. Один из простейших способов увеличить детализацию трехмерных объектов — использование техники, известной как наложение текстур. *Наложение текстур (texture mapping)* — это техника, применяемая для раскраски полигона с использованием изображения (как показано на рис 2.14), что улучшает визуальное оформление рисуемых объектов. Растровые изображения обычно называют *текстурами (texture)*, поэтому при обсуждении трехмерной визуализации я буду использовать оба эти термина как взаимозаменяемые.

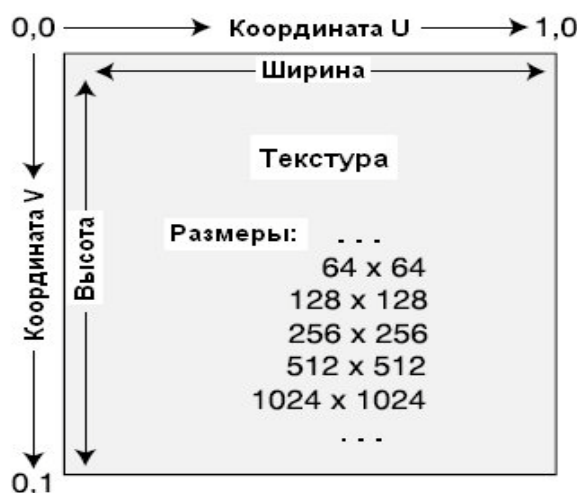


**Рис. 2.14.** При наложении текстуры вы берете плоские полигоны и рисуете на их поверхности картинку

При наложении текстур вы назначаете каждой вершине полигона пару координат. Эти координаты (называемые *координаты U, V*) определяют точку внутри изображения текстуры. Координаты U, V являются аналогом координат X, Y изображения текстуры, но вместо того, чтобы указывать координаты основанные на ширине и высоте изображения текстуры в пикселях, вы задаете координаты в диапазоне от 0.0 до 1.0.

Как правило, координаты X и Y находятся в диапазоне от нуля до ширины и высоты изображения соответственно, так что если у вас есть изображение размером  $640 \times 480$ , X будет в диапазоне от 0 до 639, а Y — в диапазоне от 0 до 479. Для доступа к пикселю, находящемуся в середине изображения, вы указываете координаты  $X = 319$  и  $Y = 239$ .

Координаты U и V находятся в диапазоне от 0 (верхний или левый край изображения) до 1 (правый или нижний край изображения), как показано на рис. 2.15. Для доступа к пикселю, находящемуся в центре изображения размером  $640 \times 480$  точек, вы используете координаты  $U = 0.5$  и  $V = 0.5$ .



**Рис. 2.15.** Координаты U и V для изображения текстуры остаются постоянными, независимо от размера изображения

На первый взгляд договоренность относительно координат U и V может показаться странной. Однако она замечательно работает, поскольку вы можете быстро заменить текстуру на новую другого размера и не беспокоиться об изменении координат.

#### СОВЕТ

Вот изящный трюк. Вы можете задавать для координат U и V значения больше, чем 1.0. В результате при визуализации текстура будет дублироваться. Например, если вы зададите для координаты U значение 2.0, текстура будет нарисована дважды, дублируясь по горизонтали. Если задать для координаты U значение 3.0, текстура будет отображена трижды. То же самое справедливо и для координаты V.

Текстурой может быть все, что хотите, но почти всегда это растровые изображения. В современных видеокартах добавлена технология наложения рельефа, которая берет текстуру и рассматривает ее как шероховатую поверхность, что делает визуализируемый трехмерный объект выглядящим так, будто на нем есть неровности.

Использование наложения рельефа достаточно сложно для текущих целей, так что, желая оставить материал простым, я покажу вам только как наложить изображение текстуры на поверхность полигона, чтобы улучшить визуальное представление в вашей графической системе.

## Использование наложения текстур в Direct3D

Управление текстурами в Direct3D осуществляется через объект **IDirect3DTexture9**. Этот объект хранит информацию о текстуре и предоставляет доступ к ней (включая указатель на пиксели, образующие изображение текстуры).

Начав использовать текстуры, вы столкнетесь с некоторыми ограничениями, накладываемыми на них Direct3D и производителями видеокарт. Во-первых, ограничены размеры текстур, которые к тому же должны быть степенями двойки (например, 8, 32, 128 или 256). Обычно ширина текстуры равна ее высоте, то есть используются размеры  $128 \times 128$  или  $256 \times 256$ . Будьте внимательны, поскольку при работе с трехмерной графикой здесь может крыться ловушка: некоторые видеокарты не позволяют применять текстуры, у которых высота отличается от ширины (то есть, размером  $128 \times 64$  или  $32 \times 256$ ), и большинство (на момент написания книги) не разрешают использовать текстуры, размеры которых не являются степенями двойки. Поэтому вы всегда должны стараться использовать текстуры у которых ширина равна высоте. Кроме того, вы должны гарантировать, что размеры ваших текстур не превышают  $256 \times 256$ , поскольку это максимальный размер текстур с которыми могут работать большинство видеокарт (а вы должны удостовериться, что ваша игра совместима с большей частью существующего оборудования).

И, наконец, не используйте слишком много текстур. Хотя процесс визуализации полигона с наложенной текстурой является достаточно простой задачей для современных видеокарт, подготовка текстур к использованию не столь проста. Каждый раз, когда оборудованию требуется текстура, Direct3D и ваша видеокарта должны выполнить ряд действий, чтобы подготовиться к текстурированию.

В эти действия входит копирование текстуры в соответствующую область памяти (если ее нет там) и установка формата цвета, чтобы он соответствовал установленному видеорежиму (а также и внутреннему режиму представления цвета). Эти процессы отнимают время у системы и видеокарты, и чем меньше вы пользуетесь ими, тем лучше.

---

### ПРИМЕЧАНИЕ

Большинство видеокарт поддерживают прямой доступ к памяти или быструю передачу данных, что ускоряет загрузку текстур в видеопамять. Можно спокойно предполагать, что большинство видеокарт используют интерфейс AGP, поддерживающий сверхбыструю передачу данных, а это значит, что ваши текстуры и другие графические данные будут загружаться в память видеокарты настолько быстро, насколько возможно.

---

**СОВЕТ**

Чтобы уменьшить время инициализации, затрачиваемое видеокартой на подготовку текстуры, вы можете упаковать несколько изображений в одну текстуру. Сделав так, вы гарантируете, что подготовку текстуры необходимо будет выполнить только один раз; потом, по мере необходимости, отдельные изображения будут извлекаться из текстуры. Пример использования этой техники будет показан в книге.

## Загрузка текстуры

Чтобы получить изображение текстуры, вы обычно загружаете его с диска или из другого ресурса. Фактически, библиотека D3DX содержит ряд функций для загрузки и управления текстурами значительно облегчающих эту работу. Функции загрузки текстур из библиотеки D3DX перечислены в таблице 2.5.

**Таблица 2.5.** Функции загрузки текстур библиотеки D3DX

Функция	Описание
<code>D3DXCreateTextureFromFile</code>	Загружает текстуру из файла с растровым изображением
<code>D3DXCreateTextureFromFileEx</code>	Более сложная версия функции <code>D3DXCreateTextureFromFile</code>
<code>D3DXCreateTextureFromFileInMemory</code>	Загружает текстуру из файла, который уже загружен в память
<code>D3DXCreateTextureFromFileInMemoryEx</code>	Более сложная версия функции <code>D3DXCreateTextureFromFileInMemory</code>
<code>D3DXCreateTextureFromResource</code>	Загружает изображение текстуры из ресурсов приложения
<code>D3DXCreateTextureFromResourceEx</code>	Более сложная версия функции <code>D3DXCreateTextureFromResource</code>

Как видно в таблице 2.5, у каждой функции загрузки текстур есть две версии: одна быстрая и простая версия для загрузки текстур, и другая, с суффиксом **Ex**, более сложная, которая предоставляет больше контроля над процессом создания текстуры. Одиссею с наложением текстур мы начнем со знакомства с первой и наиболее простой для использования функцией **D3DXCreateTextureFromFile**:

```
HRESULT D3DXCreateTextureFromFile(
    IDirect3DDevice9 *pDevice,          // Ранее инициализированный
                                         // объект устройства
    LPCSTR pSrcFile,                   // Имя загружаемого файла
                                         // с изображением
    IDirect3DTexture9 **ppTexture);    // Создаваемый объект текстуры
```

И снова с функцией не сложно иметь дело; просто передайте ей ранее инициализированный объект устройства, имя файла с изображением,

который вы хотите загрузить, и указатель на созданный вами объект **IDirect3DTexture9**.

Вот пример использования функции **D3DXCreateTextureFromFile** для загрузки растрового изображения из файла с именем texture.bmp в объект текстуры:

```
// g_pD3DDevice = ранее инициализированный объект устройства
IDirect3DTexture9 *pD3DTexture;

if (FAILED(D3DXCreateTextureFromFile(g_pD3DDevice,
                                     "texture.bmp", (void**)&pD3DTexture))) {
    // Произошла ошибка
}
```

Самое лучшее в этой функции то, что она выполняет всю инициализацию за вас и помещает структуру в память **D3DPOOL\_MANAGED**, а это значит, что текстура будет постоянно находиться в памяти (потерянные текстуры были главной головной болью для программистов до выхода DirectX 8).

## Установка текстуры

Как говорилось раньше, в разделе «Использование наложения текстур в Direct3D», видеокарта должна подготовиться к использованию текстуры для визуализации. Эта подготовка должна быть выполнена перед тем, как будет визуализироваться полигон с текстурой. Если у вас 1000 полигонов и для каждого полигона используется своя текстура, вы должны в цикле перебирать полигоны, устанавливая для каждого полигона его текстуру и визуализировать его.

Вы повторяете этот процесс, пока не будут визуализированы все полигоны. Если несколько полигонов используют одну и ту же текстуру, более эффективно установить текстуру и визуализировать все полигоны, которые используют ее, вместо того, чтобы использовать цикл установки текстуры и визуализации для каждого полигона.

Для установки текстуры используют функцию **IDirect3DDevice9::SetTexture**:

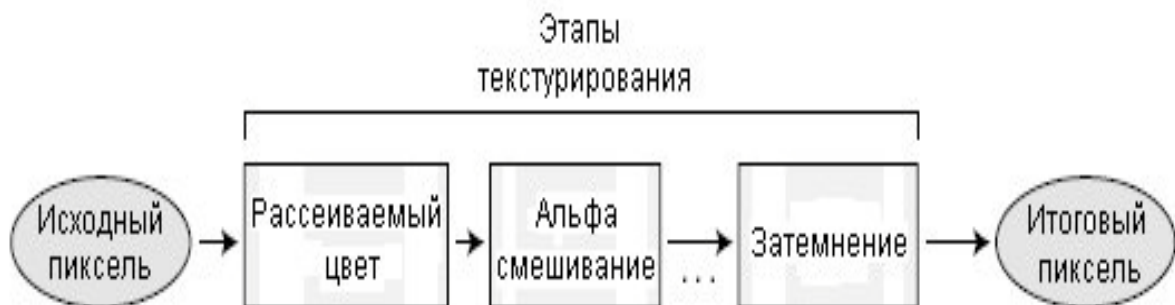
```
HRESULT IDirect3DDevice9::SetTexture(
    DWORD          Stage,          // Этап текстурирования 0 - 7
    IDirect3DBaseTexture9 *pTexture); // Устанавливаемый объект текстуры
```

Вы видите, где передается созданный вами объект текстуры (как **pTexture**), но что за параметр **Stage**? Он называется *этапом текстурирования* (*texture stage*) и это одна из самых захватывающих техник наложения текстур в Direct3D.

Процесс наложения текстур в Direct3D очень гибкий. Текстура не берется из единственного источника, а может быть построена с использованием целых восьми различных источников. Эти источники, называемые *этапами текстурирования* (*texture stage*), нумеруются от 0 до 7. При визуализации полигона для каждого рисуемого пикселя Direct3D

начинает с этапа 1 и запрашивает пиксель текстуры. Затем Direct3D переходит к этапу 2 и запрашивает другой пиксель текстуры или позволяет вам модифицировать пиксель предыдущей текстуры. Процесс продолжается, пока не будут пройдены все восемь этапов.

Каждый этап может изменить пиксель текстуры как требуется, включая смешивание пикселя с пикселем новой текстуры, увеличение или уменьшение насыщенности цвета или яркости и даже выполнение специального эффекта, известного как альфа-смешивание (техника, которая смешивает цвета нескольких пикселей). Этот процесс изображен на рис. 2.16, где исходный пиксель проходит через каждый этап, начиная с этапа 0, на котором из текстуры извлекается рассеиваемая составляющая цвета пикселя (уровни красного, зеленого и синего цветов). Затем для пикселя выполняется альфа-смешивание, после чего производится затемнение, и в результате получается итоговый пиксель, который рисуется на экране.



**Рис. 2.16.** Каждый этап текстурирования модифицирует исходный пиксель различными способами. Здесь исходный пиксель проходит через ряд изменений для получения итогового пикселя

Возможности этапов текстурирования бесконечны, но, к сожалению, у меня нет места, чтобы подробно обсудить их. В этой книге я использую единственный этап текстурирования, который извлекает цвет пикселя из текстуры, применяет информацию о цвете полигона и отображает полученный окрашенный пиксель на полигон.

Приведенный ниже фрагмент кода устанавливает текстуру для использования на этапе 0 и сообщает визуализатору о необходимости извлечь пиксель текстуры, применить информацию о цветах вершин и запретить альфа-смешивание:

```
// g_pD3DDevice = ранее инициализированный объект устройства
// pD3DTexture = загруженный объект текстуры

// Устанавливаем текстуру для этапа 0
g_pD3DDevice->SetTexture(0, pTexture);

// Устанавливаем параметр этапа - это необходимо делать
// один раз в программе
g_pD3DDevice->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE);
g_pD3DDevice->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
g_pD3DDevice->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
g_pD3DDevice->SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_DISABLE);
```

Это основной набор операций с текстурами и вы будете встречать его достаточно часто. Обратите внимание, что вы устанавливаете параметры этапа текстурирования только один раз, и затем полагаетесь на вызов **SetTexture**. Чтобы узнать больше об использовании этапов текстурирования, обратитесь к документации DX SDK.

Когда вы закончили использовать текстуру (после визуализации полигонов), вы вызываете функцию **SetTexture** еще раз, указав в параметре **pTexture** значение **NULL**:

```
g_pD3DDevice->SetTexture(0, NULL);
```

Этот вызов удаляет данные текстуры из памяти и видеопроцессора. Если не сделать этого, в программе появятся утечки памяти, которые могут привести даже к краху вашей игры.

## Использование выборок

Время от времени вы будете встречать упоминание *режимов выборки* (*sampler state*). Режимы выборки вступают в игру, когда визуализируются полигоны с текстурами. Поскольку разрешение экрана конечно, на изображении возникают небольшие визуальные аномалии, такие как зазубренные края у наклонных линий или пикселизация изображения текстуры при его масштабировании.

По этим (и по многим другим) причинам были созданы выборки, сглаживающие эти небольшие дефекты. Direct3D использует различные выборки, которые гарантируют, что ваша графика будет выглядеть более чисто.

Чтобы использовать выборку в Direct3D необходимо воспользоваться функцией **IDirect3DDevice9::SetSamplerState**:

```
HRESULT IDirect3DDevice9::SetSamplerState(  
    DWORD Sampler, // Этап текстурирования/выборки 0 - 7  
    D3DSAMPLERSTATETYPE Type, // Устанавливаемый режим  
    DWORD Value); // Используемое значение
```

В этой книге я буду задавать только две выборки: **D3DSAMP\_MINFILTER** и **D3DSAMP\_MAGFILTER**. Оба этих состояния определяют как Direct3D смешивает близлежащие пиксели внутри текстуры перед выводом пикселя на экран. Первое состояние, **D3DSAMP\_MAGFILTER**, используется когда текстура растягивается на полигон (увеличивается), а **D3DSAMP\_MINFILTER** — когда текстура сжимается (уменьшается).

Аргумент **Value** может принимать одно из значений, перечисленных в таблице 2.6.



Таблица 2.6. Типы фильтров выборки текстуры в Direct3D

Значение	Описание
<b>D3DTEXF_NONE</b>	Фильтр не используется
<b>D3DTEXF_POINT</b>	Самый быстрый режим фильтрации. Используется цвет единственной точки из текстуры
<b>D3DTEXF_LINEAR</b>	Режим билинейной интерполяции. Этот режим для получения итогового смешанного пикселя комбинирует четыре пикселя текстуры. Достаточно быстрый режим наложения текстуры, позволяющий получить сглаженное изображение
<b>D3DTEXF_ANISOTROPIC</b>	Анизотропная фильтрация компенсирует угловые различия между экраном и текстурируемым полигоном. Хорошо, но медленно

Как правило, вы будете использовать режимы фильтрации **D3DTEXF\_POINT** или **D3DTEXF\_LINEAR**; они быстрые, а линейный режим еще и сглаживает результат. Чтобы использовать какой-нибудь режим фильтрации просто добавьте следующий код:

```
// g_pD3DDevice = ранее инициализированный объект устройства
// Установка фильтра увеличения
if (FAILED(g_pD3DDevice->SetSamplerState(0,
    D3DSAMP_MAGFILTER, D3DTEXF_POINT))) {
    // Произошла ошибка
}

// Установка фильтра уменьшения
if (FAILED(g_pD3DDevice->SetSamplerState(0,
    D3DTSS_MINFILTER, D3DTEXF_POINT))) {
    // Произошла ошибка
}
```

## Визуализация текстурированных объектов

Перед тем, как вы сможете рисовать объект (полигон или набор полигонов) с текстурой, вы должны убедиться, что в данные вершин полигона включена пара координат U, V. Структура данных вершины, содержащая только набор трехмерных координат и координаты текстуры, выглядит так:

```
typedef struct {
    D3DVECTOR3 Position; // Вектор местоположения вершины
    float      tu, tv;    // Здесь добавляем координаты текстуры!
} sVertex;
```

Теперь вы конструируете макроопределение настраиваемого формата вершин, чтобы информировать Direct3D о том, какие компоненты данных вершины используются. В данном случае компоненты — это непреобразованные трехмерные координаты и пара координат текстуры.

В описании используются значения **D3DFVF\_XYZ** и **D3DFVF\_TEX1**:

```
#define VERTEXFMT (D3DFVF_XYZ | D3DFVF_TEX1)
```

Теперь самая интересная часть — помещение вашей графики на экран. Добавив несколько строк кода вы можете усовершенствовать простую функцию рисования полигона для включения вашей текстуры. Вот пример загрузки текстуры и использования ее при рисовании списка треугольников (подразумевается, что вы уже инициализировали устройство, определили буфер вершин с информацией о текстуре и установили матрицы мирового преобразования, преобразования вида и проекции):

```
// g_pD3DDevice = ранее инициализированный объект устройства
// NumPolys      = количество рисуемых полигональных примитивов
// g_pD3DVertexBuffer = ранее созданный буфер вершин
//                  с информацией о полигонах

IDirect3DTexture9 *pD3DTexture;      // Объект текстуры

// Загрузка текстуры
D3DXCreateTextureFromFile(g_pD3DDevice, "texture.bmp",
                          (void**) &pD3DTexture);

if(SUCCEEDED(g_pD3DDevice->BeginScene())) {

    // Установка текстуры
    g_pD3DDevice->SetTexture(0, pD3DTexture);

    // Установка источника потоковых данных
    // и вершинного шейдера
    g_pD3DDevice->SetStreamSource(0, g_pD3DVertexBuffer, 0,
                                   sizeof(sVertex));
    g_pD3DDevice->SetFVF(VERTEXFMT);

    // Рисуем список треугольников
    g_pD3DDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 0, NumPolys);

    // Завершение сцены
    g_pD3DDevice->EndScene();

    // Освобождение ресурсов текстуры
    g_pD3DDevice->SetTexture(0, NULL);
}
```

## Альфа-смешивание

Вообразите, что вы находитесь в одном из самых высоких в мире зданий, подходите к окну и смотрите на расстилающийся внизу город. Легкий синеватый оттенок стекла придает картине мирный вид, подобный утреннему небу.

Представьте ту же сцену еще раз, но теперь на языке трехмерной графики. Весь мир создан из полигонов, которые для всех практических целей считаются сплошными объектами. Вы не можете смотреть сквозь них.

Но что, если вы хотите, чтобы в вашей игре были прозрачные окна? И как насчет замечательного оттенка, который окно придает картине?

Вы хотите получить не только замечательные эффекты, подобные упомянутым выше, но и такие вещи, как *копирование с учетом прозрачности* (*transparent blits*) (то есть вы хотите рисовать полигоны отдельные фрагменты которых будут полностью прозрачными).

Думайте о рисовании частично прозрачных объектов как о стене с дырой в центре. Дыра полностью прозрачна, хотя сама стена является сплошной; вы можете беспрепятственно смотреть сквозь отверстие.

Такие эффекты можно реализовать с помощью техники называемой *альфа-смешивание* (*alpha blending*). Используя альфа-смешивание вы можете менять прозрачность полигонов, чтобы сквозь них можно было видеть. Если полигон окрашен, то его цвет будет смешиваться с цветами находящихся позади него объектов. Более того, вы можете даже использовать текстуру для полигона, чтобы создать ошеломляющие эффекты!

Степень прозрачности объекта называется *альфа-значением* (*alpha value*). Как вы, возможно, уже заметили, Direct3D использует альфа-значение различными способами. Например, используя текстуры вы можете задать формат в котором есть альфа-значения. Хранилище альфа-значений называется *альфа-каналом* (*alpha channel*).

#### ПРИМЕЧАНИЕ

*Альфа-канал* (*alpha channel*) — это значение, точно такое же, как цветовые компоненты (красный, зеленый и синий). Оно определяет степень прозрачности для каждого пикселя поверхности, имеющего собственный альфа-канал.

Альфа-канал может занимать от одного до восьми битов. Если у вас 8-разрядный альфа-канал, вы можете задать 256 альфа-значений (в диапазоне от 0 до 255). Четырехразрядный альфа-канал позволяет использовать 16 альфа-значений (от 0 до 15).

## Включение альфа-смешивания

Включение альфа-смешивания в Direct3D осуществляется простой установкой необходимых режимов визуализации с помощью функции **IDirect3DDevice9::SetRenderState**. Первый режим визуализации, который собственно и включает альфа-смешивание, это **D3DRS\_ALPHABLENDENABLE**:

```
// g_pD3DDevice = ранее инициализированный объект устройства
// Для включения альфа-смешивания используйте:
g_pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);

// Установка типа альфа-смешивания
g_pD3DDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
g_pD3DDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);

// Для выключения альфа-смешивания используйте:
g_pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, FALSE);
```

Обратите внимание на два дополнительных режима визуализации (**D3DRS\_SRCBLEND** и **D3DRS\_DESTBLEND**) в приведенном выше коде. Они сообщают Direct3D как надо использовать указанные альфа-значения при визуализации. Время от времени вы будете видеть, что для режима **D3DRS\_DESTBLEND** устанавливается значение **D3DBLEND\_ONE**, а не **D3DBLEND\_INVSRCALPHA**. Я буду специально обращать ваше внимание на такие случаи.

## Рисование с альфа-смешиванием

Чтобы использовать альфа-смешивание вам необходима информация о том, как добавить альфа-значения к вашему формату вершин. Это осуществляется путем добавления рассеиваемой компоненты цвета к структуре данных вершины и дескриптору. Когда вы задаете рассеиваемый цвет, вы можете также указать и альфа-значение.

Приведенный ниже пример демонстрирует простую структуру данных вершины, хранящую трехмерные координаты и рассеиваемую составляющую цвета (которая теперь включает альфа-значение):

```
// Структура данных вершины и дескриптор
typedef struct {
    FLOAT      x, y, z;
    D3DCOLOR diffuse;
} sVertex;
#define VertexFVF (D3DFVF_XYZ | D3DFVF_DIFFUSE)

// Определяем массив из трех вершин
sVertex Verts = {
    { 0.0f, 100.0f, 0.0f, D3DCOLOR_RGBA(255, 0, 0, 64) },
    { 100.0f, -100.0f, 0.0f, D3DCOLOR_RGBA( 0,255, 0,128) },
    { -100.0f, -100.0f, 0.0f, D3DCOLOR_RGBA( 0, 0,255,255) }
};
```

Для первой вершины задан красный цвет и прозрачность 1/4 (1/4 цвета будет смешиваться). Вторая вершина зеленая с прозрачностью 1/2 (1/2 цвета будет смешиваться). Третья вершина синяя и полностью непрозрачна, а это значит, что для нее цвета не будут смешиваться.

Если вы добавляете координаты текстуры и устанавливаете текстуру, можно установить максимальные значения цветовых компонент (255 для зеленого, красного и синего) и затем задать альфа-значение для смешивания текстуры.

## Копирование с учетом прозрачности и альфа-проверка

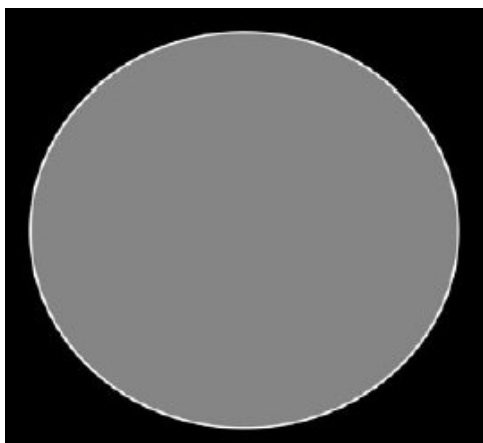
*Альфа-проверка (alpha testing)* — это техника, проверяющая альфа-значения пикселей перед их рисованием на экране. Те пиксели, альфа-значения которых не попадают в заданный диапазон, отвергаются и, следовательно, не влияют на результат визуализации. Подобно способу, которым в предыдущем разделе вы достигали эффекта полупрозрачности, вы можете

использовать альфа-проверку для визуализации полигонов с полностью прозрачными фрагментами.

Воспользовавшись приведенным ранее примером «дырка в стене», вообразите, что стена — это полигон и вы хотите нарисовать отверстие в ее центре. Вам необходимо, чтобы полигон был полностью непрозрачным (сплошным), за исключением отверстия. При этом вы хотите, чтобы отверстие было полностью прозрачным. Чтобы достичь такого эффекта, используется техника, называемая *копирование с учетом прозрачности* (*transparent blit*), которая позволяет вам исключать части текстуры при визуализации и, таким образом, видеть находящиеся позади объекты сквозь эти исключенные фрагменты.

Секрет копирования с учетом прозрачности заключается в установке вашей текстуры и использованию одного из ее цветов в качестве цветового ключа. *Цветовой ключ* (*color key*) — это цвет, который не рисуется при визуализации полигона.

Например, если у вас есть текстура на которой изображен круг в центре, окруженный черным цветом (как показано на рис. 2.17), вы можете установить в качестве цветового ключа черный цвет. Когда текстура будет применена к полигону, а сам полигон нарисован, Direct3D не будет рисовать черные пиксели, так что отображен будет только круг в центре.



*Рис. 2.17. Текстура с кругом в центре может использовать копирование с учетом прозрачности для исключения темной внешней области*

В действительности, пиксели отмечаются как прозрачные не с помощью цветового ключа, а с помощью альфа-значения пикселя. Чтобы пиксель был полностью прозрачным, его альфа-значение должно быть равно нулю. Для пикселя, который будет нарисован, альфа-значение должно быть наивысшим, обычно 255.

Как вы можете предположить, пиксели, соответствующие цветовому ключу имеют альфа-значение 0; а для всех остальных используется наибольшее альфа-значение.

## Загрузка текстур с цветовым ключом

Когда альфа-проверка используется рассматриваемым способом, вы не указываете рассеиваемую составляющую цвета в структуре данных вершины или дескрипторе. Альфа-значения хранятся непосредственно в данных

пикселей текстуры. Чтобы установить альфа-значения в данных пикселей текстуры вы загружаете текстуру используя расширенную версию функции **D3DXCreateTextureFromFile**, прототип которой выглядит так:

```
HRESULT D3DXCreateTextureFromFileEx(
    LPDIRECT3DDEVICE9 pDevice,          // Ранее созданный объект устройства
    LPCSTR pSrcFile,                   // Имя файла с загружаемой текстурой
    UINT Width,                        // D3DX_DEFAULT
    UINT Height,                       // D3DX_DEFAULT
    UINT MipLevels,                    // D3DX_DEFAULT
    DWORD Usage,                       // 0
    D3DFORMAT Format,                   // Используемый формат цвета
    D3DPOOL Pool,                      // D3DPOOL_MANAGED
    DWORD Filter,                       // D3DX_FILTER_TRIANGLE
    DWORD MipFilter,                   // D3DX_FILTER_TRIANGLE
    D3DCOLOR ColorKey,                 // Используемый цветовой ключ!
    D3DXIMAGE_INFO *pSrcInfo,          // NULL
    PALETTEENTRY *pPalette,            // NULL
    LPDIRECT3DTEXTURE9 *ppTexture);    // Созданный объект текстуры
```

Для большинства параметров указываются представленные значения по умолчанию. Вам необходимо предоставить только имя файла с растровым изображением для загрузки, указатель на объект устройства, который будет использоваться для создания текстуры, формат цвета, используемый при загрузке текстуры (тип **D3DFMT\_\*** в котором используется альфа-значение, такой как **D3DFMT\_A8R8G8B8**) и цветовой ключ (в формате **D3DCOLOR**).

Для указания значения цветового ключа, можете использовать макросы **D3DCOLOR\_RGBA** или **D3DCOLOR\_COLORVALUE**. Например, если вы хотите исключить из процесса рисования черный цвет, при загрузке текстуры используйте следующее значение цветового ключа:

```
D3DCOLOR_RGBA(0, 0, 0, 255);
```

Обратите внимание на значение 255 для альфа. Это очень важно! При загрузке файла формата .BMP вы должны указывать для альфа-значения число 255. Если вы работаете с другими форматами файлов (такими, как .TGA), которые уже содержат значения альфа-канала, необходимо, чтобы альфа-значение цветового ключа соответствовало альфа-значению, которое уже хранится в файле изображения.

В этой книге я буду использовать только файлы формата .BMP, так что просто помните о необходимости использовать альфа-значение 255. После установки альфа-значения для каждого пикселя текстуры, остается только воспользоваться альфа-проверкой, чтобы исключить пиксели на основании их альфа-значений.

## Включение альфа-проверки

Когда текстура загружена (и цветовой ключ и альфа-значения установлены) вы должны включить альфа-проверку, добавив следующие строки в код инициализации или в цикл визуализации:

```
// g_pD3DDevice = ранее инициализированный объект устройства
g_pD3DDevice->SetRenderState(D3DRS_ALPHATESTENABLE, TRUE);
g_pD3DDevice->SetRenderState(D3DRS_ALPHAREF, 0x08);
g_pD3DDevice->SetRenderState(D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL);
```

Главный волшебник здесь — режим **D3DRS\_ALPHAREF**, поскольку именно он говорит Direct3D, какие альфа-значения допустимы (в диапазоне от 0 до 255). После выполнения трех показанных вызовов функций будут отвергаться все пиксели, у которых альфа-значение меньше 8. Если вы правильно установили ваш цветовой ключ, три вызова функций приведут к тому, что все фрагменты текстур, у которых альфа-значение равно нулю будут исключены из стадии визуализации, то есть станут прозрачными!

## Пример копирования с учетом прозрачности

Достаточно разговоров; пришло время для кода! Вот небольшой пример, который загружает изображение кнопки и выводит его на экран. Черные пиксели текстуры исключаются, позволяя видеть сквозь них цвет фона.

```
// g_pD3DDevice = ранее инициализированный объект устройства

// Структура данных вершины и дескриптор
typedef struct {
    FLOAT x, y, z, rhw; // Экранные координаты
    FLOAT u, v;         // Координаты текстуры
} sVertex;
#define VertexFVF (D3DFVF_XYZRHW | D3DFVF_TEX1)

// Буфер вершин и текстура
IDirect3DVertexBuffer9 *g_pVB      = NULL;
IDirect3DTexture9      *g_pTexture = NULL;

// Инициализация буфера вершин и текстуры
// подразумевает, что размер окна 400 x 400
BYTE *Ptr;
sVertex Verts[4] = {
    { 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f },
    { 399.0f, 0.0f, 0.0f, 1.0f, 1.0f, 0.0f },
    { 0.0f, 399.0f, 0.0f, 1.0f, 0.0f, 1.0f },
    { 399.0f, 399.0f, 0.0f, 1.0f, 1.0f, 1.0f }
};

// Создание буфера вершин и заполнение данными
g_pD3DDevice->CreateVertexBuffer(sizeof(sVertex) * 4, 0,
    VertexFVF, D3DPOOL_MANAGED, &g_pVB, NULL)
g_pVB->Lock(0,0, (void*)&Ptr, 0))
memcpy(Ptr, Verts, sizeof(Verts));
g_pVB->Unlock();

// Получение текстуры
D3DXCreateTextureFromFileEx(g_pD3DDevice, "button.bmp",
    D3DX_DEFAULT, D3DX_DEFAULT, D3DX_DEFAULT, 0,
    D3DFMT_A8R8G8B8, D3DPOOL_MANAGED, D3DX_FILTER_TRIANGLE,
    D3DX_FILTER_TRIANGLE, D3DCOLOR_RGBA(0,0,0,255), NULL,
    NULL, &g_pTexture);

// Установка альфа-проверки
g_pD3DDevice->SetRenderState(D3DRS_ALPHATESTENABLE, TRUE);
g_pD3DDevice->SetRenderState(D3DRS_ALPHAREF, 0x01);
```

```

g_pD3DDevice->SetRenderState(D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL);

// Очистка вторичного буфера устройства
g_pD3DDevice->Clear(0, NULL, D3DCLEAR_TARGET,
                  D3DCOLOR_RGBA(0,128,128,255), 1.0f, 0);

if(SUCCEEDED(g_pD3DDevice->BeginScene())) {

    // Установка буфера вершин в качестве источника потоковых данных
    g_pD3DDevice->SetStreamSource(0, g_pVB, 0, sizeof(sVertex));

    // Установка вершинного шейдера
    g_pD3DDevice->SetFVF(VertexFVF);

    // Установка текстуры
    g_pD3DDevice->SetTexture(0, g_pTexture);

    // Рисование буфера вершин
    g_pD3DDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
    g_pD3DDevice->EndScene();

    // Очистка текстуры
    g_pD3DDevice->SetTexture(0, NULL);

    // Выключение альфа-проверки
    g_pD3DDevice->SetRenderState(D3DRS_ALPHATESTENABLE, FALSE);
}

// Переключаем поверхности для отображения результата
g_pD3DDevice->Present(NULL, NULL, NULL, NULL);

```

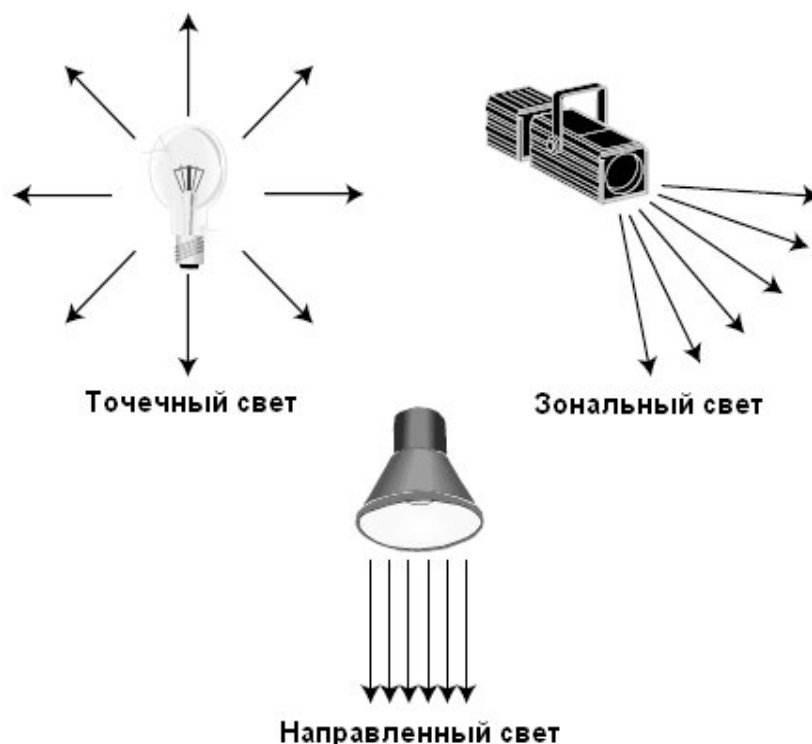
## Освещение

Следующим в списке профессиональных графических техник идет использование освещения. В отличие от реальной жизни в большинстве игр сцена полностью освещена, что делает графику четкой, но нереалистичной. Чтобы получить более реалистичную сцену и добавить к вашей графике те тонкие световые эффекты, которые своят с ума игроков, необходимо воспользоваться предоставляемыми Direct3D возможностями освещения. В Direct3D можно использовать четыре типа освещения: фоновое, точечное, направленное и зональное.

*Фоновый свет (ambient light)* — это постоянный источник света, который освещает все объекты сцены с одинаковой интенсивностью. Являясь частью аппаратуры, фоновый свет является единственным типом освещения, обрабатываемым отдельно от процесса расчета освещения.

Другие три типа источников света (изображенные на рис. 2.18) обладают уникальными свойствами. *Точечный источник света (point light)* освещает все вокруг (подобно электрической лампочке). *Зональный свет (spotlight)* распространяется в указанном направлении и испускает луч конической формы. Все внутри конуса освещено, а объекты находящиеся вне конуса не освещены. *Источник направленного света (directional light)* является упрощенной версией зонального света и просто испускает лучи света в указанном направлении.





*Рис. 2.18. Точечный, зональный и направленный источники света испускают световые лучи по-разному*

Источники света размещаются на сцене точно так же, как и остальные трехмерные объекты — с помощью координат  $x$ ,  $y$  и  $z$ . У некоторых источников света, таких как зональный свет, есть вектор направления, определяющий куда они направлены. У каждого источника света есть интенсивность, диапазон действия, коэффициент затухания и цвет. Верно, с Direct3D возможны даже цветные источники света!

За исключением фонового света, все остальные источники света используют для хранения своей уникальной информации структуру **D3DLIGHT9**. Определение этой структуры выглядит так:

```
typedef struct _D3DLIGHT9 {
    D3DLIGHTTYPE Type;           // Тип источника света
    D3DCOLORVALUE Diffuse;       // Рассеиваемая составляющая цвета
    D3DCOLORVALUE Specular;      // Отражаемая составляющая цвета
    D3DCOLORVALUE Ambient;       // Фоновая составляющая цвета
    D3DVECTOR Position;          // Местоположение источника
    D3DVECTOR Direction;         // Направление испускаемых лучей
    float Range;                 // Дальность
    float Falloff;               // Затухание зонального света
    float Attenuation0;          // Коэффициент затухания 0
    float Attenuation1;          // Коэффициент затухания 1
    float Attenuation2;          // Коэффициент затухания 2
    float Theta;                 // Угол внутреннего конуса
    float Phi;                   // Угол внешнего конуса
} D3DLIGHT9;
```

Ничего себе! Большая штукавина, но она содержит всю информацию, необходимую для описания источника света. Хотя источник света и не

обязан использовать каждую переменную в структуре **D3DLIGHT9**, все источники света используют несколько полей.

Первая переменная, которую вы будете устанавливать, — **Type**, определяющая тип используемого источника света. Она может принимать значения **D3DLIGHT\_POINT** для точечного источника света, **D3DLIGHT\_SPOT** для зонального источника света или **D3DLIGHT\_DIRECTIONAL** для направленного источника света.

Следующие строки задают цвет освещения. Наиболее часто вы будете использовать поле **Diffuse**; оно определяет цвет испускаемого источником света. Обратите внимание, что отвечающие за цвет поля хранятся в формате **D3DCOLORVALUE**, который представляет собой следующую структуру:

```
typedef struct _D3DCOLORVALUE {
    float r;    // Красная составляющая (от 0.0 до 1.0)
    float g;    // Зеленая составляющая (от 0.0 до 1.0)
    float b;    // Синяя составляющая (от 0.0 до 1.0)
    float a;    // Альфа-значение (не используется)
} D3DCOLORVALUE;
```

Вы устанавливаете каждую цветовую составляющую в структуре, присваивая ей значение в диапазоне от 0.0 (выключено) до 1.0 (максимальная интенсивность). Для красного света значения будут **r** = 1.0, **g** = 0.0, **b** = 1.0, а для белого — **r** = 1.0, **g** = 1.0, **b** = 1.0. Поскольку мы имеем дело со светом, альфа-значение здесь не используется. Поля **Specular** и **Ambient** в структуре **D3DLIGHT9** определяют цвет бликов и фоновый цвет, соответственно. Вы можете спокойно присвоить в обоих полях каждой цветовой составляющей значение 1.0 (за исключением **Specular**, где вы должны устанавливать значения 0.0, если не хотите использовать блики).

#### СОВЕТ

Вы можете использовать цветовые уровни света не только для того, чтобы освещать объект, но и чтобы затемнять его. Вместо положительных значений цветовых компонентов используйте отрицательные, и посмотрите на результат!

Как я упоминал ранее, каждый источник света размещается в сцене с помощью координат *X*, *Y* и *Z* (мировых координат), которые хранятся в векторе **Position**. Еще один вектор, **Direction**, используется для того, чтобы указать направление световых лучей. О применении вектора направления вы узнаете в разделе «Использование зонального света».

Переменная **Range** определяет насколько далеко распространяется свет до полного затухания. (значение **Falloff** используется, чтобы указать как быстро затухает свет в промежутке между внутренним и внешним конусами зонального источника; обычно используемое значение 1.0 обеспечивает плавный переход). Объекты, находящиеся дальше заданного расстояния не будут освещены источником света.

Трио полей с коэффициентами затухания определяет как будет уменьшаться интенсивность света с увеличением расстояния от источника; обычно всем трем коэффициентам присваиваются нулевые значения. Использование остальных полей зависит от типа используемого источника света.

## Использование точечного света

Работать с точечным источником света проще всего; вы просто указываете его местоположение, цветовые составляющие и дальность. Чтобы определить точечный источник света, создайте экземпляр структуры **D3DLIGHT9** и заполните его необходимой информацией:

```
D3DLIGHT9 PointLight;

// Очищаем структуру
ZeroMemory(&PointLight, sizeof(D3DLIGHT9));

// Располагаем источник света в точке 0.0, 100.0, 200.0
PointLight.Position = D3DVECTOR3(0.0f, 100.0f, 200.0f);

// Делаем фоновую и рассеиваемую составляющие цвета белыми
PointLight.Diffuse.r = PointLight.Ambient.r = 1.0f;
PointLight.Diffuse.g = PointLight.Ambient.g = 1.0f;
PointLight.Diffuse.b = PointLight.Ambient.b = 1.0f;

// Устанавливаем дальность равной 1000 единиц
PointLight.Range = 1000.0f;
```

## Использование зонального света

Зональный источник света отличается от других, потому что испускает лучи света в форме конуса, расходящегося от источника. Свет наиболее ярк в центре и постепенно затухает к краю конуса. За пределами конуса ничего не освещается.

---

**ВНИМАНИЕ!**

Зональный источник света требует наибольшего объема вычислений, поэтому не следует использовать в сцене много источников света такого типа.

---

При определении зонального источника света задается его местоположение, направление, цветовые составляющие, дальность, затухание внутри конуса, коэффициенты затухания с увеличением расстояния и углы внутреннего и внешнего конусов. Вам не стоит беспокоиться о затухании внутри конуса и коэффициентах затухания с увеличением расстояния, но вы должны помнить об углах для обоих конусов.

Переменная **Phi** в структуре **D3DLIGHT9** определяет размер внешнего конуса. **Phi**, также как и **Theta**, представляет угол (в радианах). Чем дальше лучи света уходят от источника, тем шире становится освещаемый

ими круг. Программист задает значения этих переменных, а затем экспериментирует с ними, пока не получит желаемый результат.

Приведенный ниже код создает источник зонального света и задает для него местоположение, цвет, дальность, затухание и размер конусов:

```
D3DLIGHT9 Spotlight;

// Очистка данных
ZeroMemory(&SpotLight, sizeof(D3DLIGHT9));

// Источник света располагается в точке 0.0, 100.0, 200.0
SpotLight.Position = D3DVECTOR3(0.0f, 100.0f, 200.0f);

// Делаем фоновую и рассеиваемую составляющие цвета белыми
SpotLight.Diffuse.r = SpotLight.Ambient.r = 1.0f;
SpotLight.Diffuse.g = SpotLight.Ambient.g = 1.0f;
SpotLight.Diffuse.b = SpotLight.Ambient.b = 1.0f;

// Устанавливаем дальность
SpotLight.Range = 1000.0f;

// Задаем затухание внутри конуса
SpotLight.Falloff = 1.0f;

// Устанавливаем размеры конусов
Spotlight.Phi = 0.3488; // Внешний 20 градусов
Spotlight.Theta = 0.1744; // Внутренний 10 градусов
```

Теперь вам надо направить источник света в заданном направлении. Библиотека D3DX снова приходит на помощь с парой функций, позволяющих указать точку, на которую направлен источник зонального света (да и любой другой источник света). Первая из функций — это перегруженный конструктор объекта **D3DXVECTOR3**, позволяющий задать три координаты.

Для этих трех значений вы используете координаты мирового пространства, описывающие расстояние от начала координат. Если у вас где-нибудь в сцене есть источник зонального света, и вы хотите направить его вверх на точку, находящуюся в 500 единицах над ним, необходимо для объекта вектора задать значения  $X = 0$ ,  $Y = 500$ ,  $Z = 0$  (обратите внимание, что эти три координаты заданы относительно местоположения источника света). Значение вектора устанавливает следующий фрагмент кода:

```
D3DXVECTOR3 Direction = D3DXVECTOR3(0.0f, 500.0f, 0.0f);
```

В приведенном объявлении вектора **Direction** есть одна проблема: Direct3D ожидает нормализованный вектор, а это значит, что координаты должны быть в диапазоне от 0 до 1. Никаких проблем, вторая функция библиотеки D3DX, **D3DXVec3Normalize** сделает это за вас:

```
D3DXVECTOR3 *D3DXVec3Normalize(
    D3DXVECTOR3 *pOut, // Нормализованный вектор
    CONST D3DXVECTOR3 *pV); // Исходный вектор
```

Вы передаете исходный вектор (например тот, объявление которого было показано выше, содержащий координаты  $X = 0$ ,  $Y = 500$ ,  $Z = 0$ ) и указатель на новый вектор, а функция **D3DXVec3Normalize** преобразует координаты в значения из диапазона от 0 до 1. Новый вектор теперь содержит значение направления, которое можно использовать для поля направления источника света в структуре **D3DLIGHT9**.

Продолжая приведенный выше пример, зададим ориентацию зонального источника света, направив его вверх, нормализуем вектор и сохраним его в структуре **D3DLIGHT9**:

```
D3DXVECTOR3 Dir = D3DXVECTOR3(0.0f, 500.0f, 0.0f);
D3DXVec3Normalize((D3DXVECTOR3*)&Spotlight.Direction, &Dir);
```

## Использование направленного света

В терминах производительности источник направленного света является самым быстрым из всех типов освещения. Он освещает каждый полигон, который обращен к нему. Чтобы подготовить источник направленного света к использованию вы должны задать в полях структуры **D3DLIGHT9** направление и цветовые составляющие.

Возможно, вы недоумеваете, почему не используется вектор местоположения, но это логично. Подумайте об источнике направленного света, как о бесконечно большой реке, текущей в одном направлении. Независимо от положения объекта в реке, поток воды остается одним и тем же; только направление потока может меняться. Если использовать эту аналогию для света, вода представляет световые лучи, а направление потока воды — угол освещения. Освещаются все объекты в мире, независимо от их расположения.

Вспомнив техники для двух предыдущих типов освещения, взгляните на приведенный ниже пример, где устанавливается желтый источник света, направленный вниз на сцену:

```
D3DLIGHT9 DirLight;

// Очищаем данные источника света
ZeroMemory(&DirLight, sizeof(D3DLIGHT9));

// Делаем рассеиваемую и фоновую составляющие желтыми
DirLight.Diffuse.r = DirLight.Ambient.r = 1.0f;
DirLight.Diffuse.g = DirLight.Ambient.g = 1.0f;
DirLight.Diffuse.b = DirLight.Ambient.b = 0.0f;

D3DXVECTOR3 Dir = D3DXVECTOR3(0.0f, 500.0f, 0.0f);
D3DXVec3Normalize((D3DXVECTOR3*)&DirLight.Direction, &Dir);
```

---

**ВНИМАНИЕ!**

Вектор направления источника света должен содержать хотя бы одно отличное от нуля значение. Другими словами, вы не можете задать направление  $X = 0$ ,  $Y = 0$ ,  $Z = 0$ .

---

## Фоновый свет

Фоновый свет — это единственный тип освещения, который обрабатывается Direct3D по-другому. Direct3D применяет фоновый свет ко всем полигонам, независимо от их углов или их источников света, так что затенения не происходит. Фоновый свет — это постоянный уровень освещенности и, так же как для других типов освещения (точечного, зонального и направленного), вы можете задавать его цвет.

Уровень фонового освещения задается путем установки режима визуализации **D3DRS\_AMBIENT** и передачи ему значения **D3DCOLOR** (используйте макрос **D3DCOLOR\_COLORVALUE**, в котором уровни красного, зеленого и синего задаются числами из диапазона от 0 до 1) определяющего желаемый цвет.

```
g_pD3DDevice->SetRenderState(D3DRS_AMBIENT,
    D3DCOLOR_COLORVALUE(0.0f, Red, Green, Blue));
```

## Установка света

После того, как вы инициализировали структуру **D3DLIGHT9**, ее надо передать Direct3D с помощью функции **IDirect3DDevice9::SetLight**:

```
HRESULT IDirect3DDevice9::SetLight(
    DWORD          Index,      // Индекс устанавливаемого источника
    CONST D3DLIGHT9 *pLight); // Используемая структура D3DLIGHT9
```

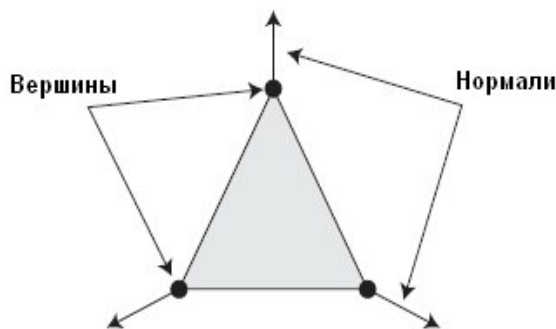
Как видите, в параметре **pLight** передается структура **D3DLIGHT9**, а в поле **Index** — что-то другое. Direct3D позволяет устанавливать на сцене несколько источников света, так что **Index** — это просто порядковый номер устанавливаемого источника света (нумерация начинается с нуля). Например, если в сцене вы используете четыре источника света, то индекс первого из них будет 0, индекс второго — 1, индекс третьего — 2 и индекс четвертого — 3.

Хотя Direct3D не накладывает ограничений на количество используемых в сцене источников света, я рекомендую стараться, чтобы их было не больше четырех. Каждый источник света увеличивает сложность сцены и время ее визуализации.

## Использование нормалей

Чтобы Direct3D мог правильно рассчитать освещение грани установленными вами источниками света, необходимо для каждой вершины полигона указать ее нормаль. *Нормаль (normal)* — это трехмерный вектор, определяющий направление в котором обращен объект (такой, как вершина или полигон) к которому этот вектор присоединен. Нормали обычно используются в сложных вычислениях, определяющих сколько света получает объект от заданного источника.

Если вы взглянете на треугольную грань (например, на ту, которая изображена на рис. 2.19), то увидите, что для трех вершин нормали задают направление. Когда свет попадает на эти вершины, угол его отражения рассчитывается на основании нормалей. Использование нормалей гарантирует, что все грани будут правильно освещены и затенение будет выполняться в соответствии с углами относительно зрителя и источника света.



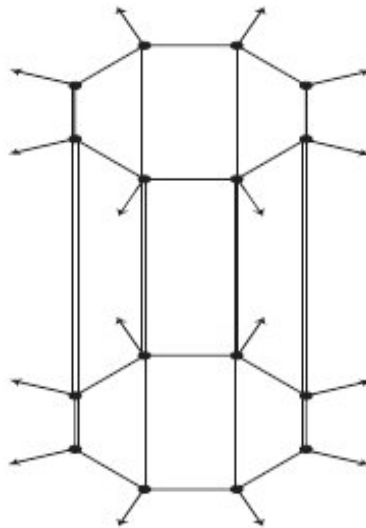
**Рис. 2.19.** У каждой вершины есть нормаль, указывающая в заданном направлении. Вы используете угол относительно нормали, чтобы определить как свет отражается от грани и как выполнять вычисления затенения

Добавить к данным вершины нормаль также просто, как и указать информацию о текстуре. Вы просто вставляете поле для нормали типа **D3DVECTOR3** и переопределяете дескриптор формата (добавив флаг **D3DFVF\_NORMAL**), как показано ниже:

```
typedef struct {
    D3DVECTOR3 Position; // Вектор координат
    D3DVECTOR3 Normal;   // Нормаль
    D3DCOLOR   Color;    // Цвет
} sVertex;
#define VERTEXFMT (D3DFVF_XYZ | D3DFVF_DIFFUSE | D3DFVF_NORMAL)
```

Нормали вы вычисляете так же, как создавали вектор направления, работая с источниками света в разделе «Использование зонального света». Когда вы начнете работать с трехмерными моделями, задача вычисления нормалей больше не будет висеть на вас, поскольку программы трехмерного моделирования, используемые для создания моделей, обычно сами вычисляют нормали.

Приведенная ниже функция (позаимствованная из примеров DirectX SDK) создает цилиндр и присваивает каждой вершине нормаль, направленную от центра цилиндра (рис. 2.20).



**Рис. 2.20.** Функция *GenerateCylinder* создает цилиндр с нормальными, направленными от центра

```
// g_pD3DDevice = ранее инициализированный объект устройства
IDirect3DVertexBuffer9 *GenerateCylinder()
{
    IDirect3DVertexBuffer9 *pD3DVertexBuffer;
    sVertex *pVertex;
    DWORD i;
    FLOAT theta;

    // Создаем буфер вершин
    if(SUCCEEDED(g_pD3DDevice->CreateVertexBuffer(
        50 * 2 * sizeof(sVertex), 0, VERTEXFMT,
        D3DPPOOL_MANAGED, &pD3DVertexBuffer, NULL))) {
        // Заполняем буфер вершин данными цилиндра
        if(SUCCEEDED(pD3DVertexBuffer->Lock(0, 0,
            (void**)&pVertex, 0))) {
            for(i = 0; i < 50; i++) {
                theta = (2 * D3DX_PI * i) / (50 - 1);
                pVertex[2*i+0].Position = D3DXVECTOR3(sinf(theta),
                    -1.0f, cosf(theta));
                pVertex[2*i+0].Normal = D3DXVECTOR3(sinf(theta),
                    0.0f, cosf(theta));
                pVertex[2*i+1].Position = D3DXVECTOR3(sinf(theta),
                    1.0f, cosf(theta));
                pVertex[2*i+1].Normal = D3DXVECTOR3(sinf(theta),
                    0.0f, cosf(theta));
            }
            pD3DVertexBuffer->Unlock();

            // Возвращаем указатель на новый буфер вершин
            return pD3DVertexBuffer;
        }
    }
    // Возвращаем NULL при ошибке
    return NULL;
}
```



## Да будет свет!

Теперь, когда вы выбрали тип источника света, который будете использовать, и инициализировали соответствующую структуру, настало время активировать конвейер освещения и включить свет. Чтобы активировать конвейер освещения надо присвоить режиму визуализации **D3DRS\_LIGHTING** значение **TRUE**:

```
// g_pD3DDevice = ранее инициализированный объект устройства
g_pD3DDevice->SetRenderState(D3DRS_LIGHTING, TRUE);
```

Чтобы выключить конвейер освещения используйте следующий код:

```
// g_pD3DDevice = ранее инициализированный объект устройства
g_pD3DDevice->SetRenderState(D3DRS_LIGHTING, FALSE);
```

После активации конвейера освещения вы включаете или выключаете отдельные источники освещения с помощью функции **IDirect3DDevice9::LightEnable**. Вот ее прототип:

```
IDirect3DDevice9::LightEnable(
    DWORD LightIndex, // Индекс источника света
    BOOL bEnable);    // TRUE для включения, FALSE для выключения
```

Если вы уже установили источник точечного света с индексом 0, то для его включения и выключения используется следующий код:

```
// g_pD3DDevice = ранее инициализированный объект устройства

// Включение источника света
g_pD3DDevice->LightEnable(0, TRUE);

// Выключение источника света
g_pD3DDevice->LightEnable(0, FALSE);
```

Вот и все, что надо для использования системы освещения Direct3D! Последнее предупреждение: Direct3D старается для освещения по максимуму использовать возможности графической системы, но если видеокарта пользователя не поддерживает освещение, Direct3D будет эмулировать световые эффекты. Это не плохо, но эмуляция может замедлить визуализацию при использовании освещения. Не позволяйте угрозе возможной эмуляции остановить вас, ведь использование световых эффектов в вашей игре значительно повышает ее привлекательность.

## Использование шрифтов

Один из недостатков DirectX 9 — слабая поддержка шрифтов. Старые версии DirectX (до версии 8) могли использовать функции вывода текста из Windows. В версии 9 вы должны вручную рисовать шрифт в поверхность текстуры и рисовать каждый символ шрифта как небольшой текстурированный полигон.

Работа с содержащей шрифт текстурой слишком сложна, чтобы использовать ее только для рисования текста, но, благодаря библиотеке D3DX, у нас есть специальный объект **ID3DXFont**, который работает с картами текстур шрифтов за вас. Объект **ID3DXFont** предоставляет семь функций, из которых нас сейчас интересуют только три.

Вот эти функции:

- **ID3DXFont::Begin** — функция отмечает начало операций рисования текста.
- **ID3DXFont::End** — функция отмечает завершение операций рисования текста.
- **ID3DXFont::DrawText** — функция рисует текст.

---

**ПРИМЕЧАНИЕ** Четвертая функция, которая может оказаться полезной — это **ID3DXFont::OnResetDevice**. Всякий раз, когда вы теряете контроль над устройством отображения (например, в тех случаях когда пользователь переключается на другое приложение) вы теряете ресурсы, такие как объект шрифта. Всякий раз, когда вы теряете ресурсы (это можно определить по коду ошибки — за дополнительной информацией обращайтесь к документации DX SDK), вам необходимо вызвать функцию **OnResetDevice** для восстановления этих ресурсов.

---

Перед тем, как двигаться дальше, давайте посмотрим как создать шрифт.

## Создание шрифта

Чтобы использовать объект **ID3DXFont** вы должны сперва инициализировать его с помощью функции **D3DXCreateFontIndirect**:

```
HRESULT D3DXCreateFontIndirect(
    IDirect3DDevice9 *pDevice, // Устройство, связанное со шрифтом
    CONST LOGFONT    *pLogFont, // Структура, описывающая шрифт
    ID3DXFont        **ppFont); // Указатель на созданный объект шрифта
```

---

**ВНИМАНИЕ!** Поскольку **ID3DXFont** — это COM-объект, не забывайте освобождать его по завершении работы.

---

Вы передаете этой функции ранее созданный объект устройства и указатель на объект **ID3DXFont**, который хотите инициализировать, но что за параметр **pLogFont**?

Как видите, **pLogFont** — это указатель на структуру **LOGFONT** (ее имя означает *логический шрифт*), которая выглядит так:

```
typedef struct tagLOGFONT {
    LONG   lfHeight;
    LONG   lfWidth;
    LONG   lfEscapement;
```

```
    LONG    lfOrientation;  
    LONG    lfWeight;  
    BYTE    lfItalic;  
    BYTE    lfUnderline;  
    BYTE    lfStrikeOut;  
    BYTE    lfCharSet;  
    BYTE    lfOutPrecision;  
    BYTE    lfClipPrecision;  
    BYTE    lfQuality;  
    BYTE    lfPitchAndFamily;  
    TCHAR    lfFaceName[LF_FACESIZE];  
} LOGFONT;
```

Как много информации! Вы можете спокойно пропустить инициализацию большинства полей в структуре **LOGFONT** и оставить для них значения по умолчанию, которые я покажу. Единственные поля, с которыми вы обязаны работать: **lfHeight**, **lfWeight**, **lfItalic**, **lfUnderline** и **lfFaceName**.

Начнем с самого простого — вы присваиваете переменным **lfItalic** и **lfUnderline** значения 0 или 1, чтобы отключить или включить использование курсива и подчеркнутого шрифта, соответственно. В переменной **lfWeight** вы задаете уровень насыщенности шрифта; 0 соответствует обычному шрифту, а 700 — полужирному. Переменная **lfHeight** представляет размер шрифта в точках. Значение **lfHeight** несколько необычно, поскольку в нем не указывается непосредственно размер. Вместо этого вы должны предоставить отрицательное число, представляющее примерную высоту шрифта в пикселях. Например, для шрифта высотой 16 пикселей вы должны указать значение -16.

И, наконец, в поле **lfFaceName** вы указываете имя шрифта, который хотите использовать. Это может быть Times New Roman, Courier New или любой другой шрифт, установленный в вашей системе. Вы просто копируете имя в поле **lfFaceName**. Вот пример в котором используется шрифт Times New Roman высотой 16 точек:

```
// g_pD3DDevice = ранее созданный объект устройства  
// hWnd         = дескриптор родительского окна  
ID3DXFont *pD3DFont;  
LOGFONT    lf;  
  
// Очищаем структуру данных шрифта  
ZeroMemory(&lf, sizeof(LOGFONT));  
  
// Устанавливаем имя шрифта и высоту  
strcpy(lf.lfFaceName, "Times New Roman");  
lfHeight = -16;  
  
// Создаем объект шрифта  
if (FAILED(D3DXCreateFontIndirect(g_pD3DDevice,  
                                  &lf, &pD3DFont))) {  
    // Произошла ошибка  
}
```

## Рисование текста

Как только ваш объект **ID3DXFont** инициализирован, вы можете начинать рисовать текст с помощью функции **ID3DXFont::DrawText**:

```
HRESULT ID3DXFont::DrawText(
    LPCSTR    pString, // Выводимая строка
    INT       Count,   // -1
    LPRECT     pRect,   // Область, в которой будет выведен текст
    DWORD      Format,  // 0
    D3DCOLOR   Color); // Цвет для рисования
```

Единственная вещь, о которой надо помнить используя функцию **DrawText**, — это параметр **pRect**, являющийся указателем на структуру **RECT**, определяющую область в которой будет нарисован текст. Вы можете установить эту область равной размерам экрана, или, если хотите, чтобы текст был только в заданной области, используйте ее экранные координаты. Вот как выглядит структура **RECT**:

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

Последний параметр функции **DrawText** — **Color**, определяет цвет, используемый для рисования текста. Для определения цвета рисуемого текста воспользуйтесь удобными макросами **D3DCOLOR\_RGBA** или **D3DCOLOR\_COLORVALUE**.

Ах да, еще один момент. Вы должны поместить ваши вызовы **DrawText** между парой вызовов **ID3DXFont::Begin** и **ID3DXFont::End**. Эти две функции информируют Direct3D о том, что вы готовитесь к визуализации текста.

В приведенном ниже примере подразумевается, что вы уже инициализировали объект шрифта и готовы выводить текст:

```
// g_pD3DDevice = ранее созданный объект устройства
// pD3DXFont     = ранее созданный объект шрифта

// Устанавливаем структуру RECT для области рисования
RECT rect = { 0, 0, 200, 100 };

// Начало блока кода рисования
if(SUCCEEDED(g_pD3DDevice->BeginScene())) {

    // Начинаем визуализацию шрифта
    pD3DXFont->Begin();

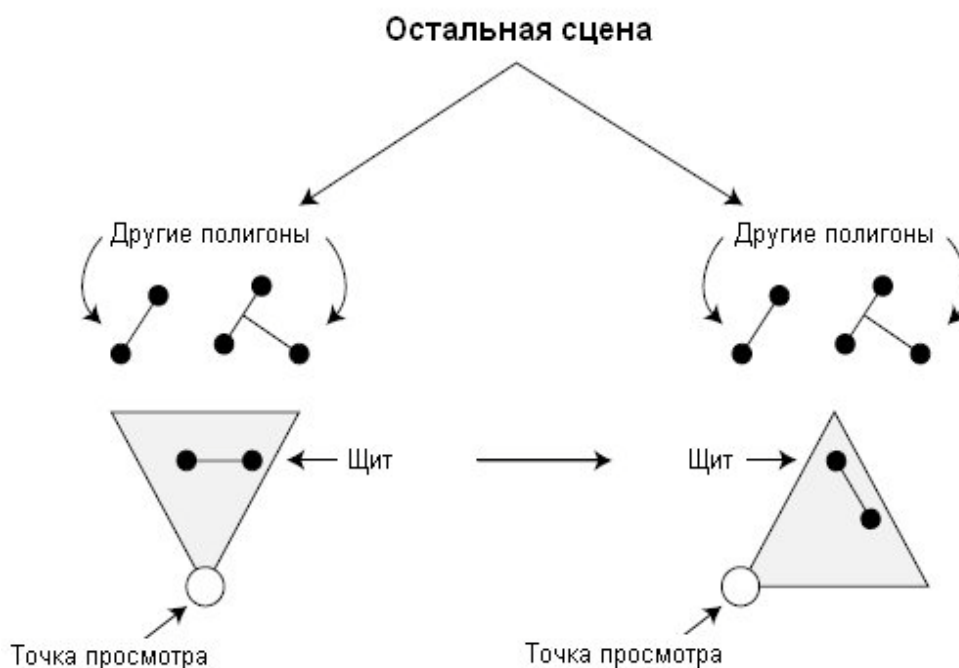
    // Рисуем какой-нибудь текст
    pD3DXFont->DrawText("I can draw with text!", -1,
        &rect, 0, D3DCOLOR_RGBA(255,255,255,255));
}
```

```
// Завершаем визуализацию шрифта
pD3DXFont->End();

// Завершаем сцену
g_pD3DDevice->EndScene();
}
```

## Щиты

*Щиты (billboard)* — это мощная техника, позволяющая двумерным объектам присутствовать в трехмерном окружении. Например, для сложного объекта, такого как дерево, в программе моделирования может быть создан вид сбоку, который потом будет накладываться как текстура на прямоугольник. Этот прямоугольник всегда будет обращен к зрителю, поэтому независимо от угла, под которым рассматривается прямоугольник, будет казаться, что текстура дерева рассматривается с той же самой точки, что и при ее создании (как показано на рис. 2.21).



**Рис. 2.21.** Щиты гарантируют, что полигон будет обращен лицевой поверхностью к зрителю, независимо от местоположения или угла, под которым смотрит зритель

Многие программисты при создании игр применяют щиты из-за простоты реализации. Замечательный пример использования щитов можно увидеть в игре Paper Mario для N64. Все персонажи нарисованы в виде двумерных изображений, а затем эти текстуры накладываются на полигоны. В игру добавлен трюк, позволяющий видеть, как вращаются вокруг полигоны щитов, что придает графике комичный стиль.

Работа щитов основана на использовании мировой матрицы, которая выравнивает полигоны относительно зрителя. Поскольку вы уже знаете углы просмотра (или можете получить матрицу преобразования вида), вам необходимо только сконструировать матрицу, используя противоположные

значения углов. Вам не надо менять местоположение полигона, поскольку важны только углы.

Первый способ построения мировой матрицы щита (которую вы можете применить к сетке или к полигону) заключается в использовании противоположных значений углов вида, которые вы уже знаете. Предположим, для примера, что буфер вершин уже заполнен данными вершин.

Углы точки просмотра хранятся в переменных **XRot**, **YRot** и **ZRot**, а координаты объекта щита — в переменных **XCoord**, **YCoord** и **ZCoord**. Вот как надо инициализировать матрицу, используемую для визуализации буфера вершин этого щита:

```
// g_pD3DDevice = ранее инициализированный объект устройства
D3DXMATRIX matBillboard;
D3DXMATRIX matBBXRot, matBBYRot, matBBZRot;
D3DXMATRIX matBBTrans;

// Конструируем матрицу щита
// Для выравнивания используем углы, противоположные
// углам просмотра
D3DXMatrixRotationX(&matBBXRot, -XRot);
D3DXMatrixRotationY(&matBBYRot, -YRot);
D3DXMatrixRotationZ(&matBBZRot, -ZRot);

// Используем координаты объекта щита для размещения
D3DXMatrixTranslation(&matBBTrans, XCoord, YCoord, ZCoord);

// Комбинируем матрицы
D3DXMatrixIdentity(&matBillboard);
D3DXMatrixMultiply(&matBillboard, &matBillboard, &matBBTrans);
D3DXMatrixMultiply(&matBillboard, &matBillboard, &matBBZRot);
D3DXMatrixMultiply(&matBillboard, &matBillboard, &matBBYRot);
D3DXMatrixMultiply(&matBillboard, &matBillboard, &matBBXRot);

// Устанавливаем матрицу
g_pD3DDevice->SetTransform(D3DTS_WORLD, &matBillboard);

// Далее можно рисовать буфер вершин, который будет выровнен
// относительно точки просмотра, с правильными координатами
```

После последней строки кода матрица мирового преобразования установлена и готова к использованию при визуализации объекта щита.

Вторым способом создания мировой матрицы щита является получение от Direct3D текущей матрицы вида и ее транспонирование (инвертирование). Эта транспонированная матрица выравнивает все относительно зрителя требуемым образом. Вы просто применяете матрицу перемещения сетки, чтобы сетка находилась в требуемом месте вашего мира. Затем вы конструируете матрицу щита из матрицы вида и используете ее для рисования объекта щита:

```
// g_pD3DDevice = ранее инициализированный объект устройства
D3DXMATRIX matTrans, matWorld, matTransposed;

// Получаем текущую матрицу вида Direct3D
g_pD3DDevice->GetTransform(D3DTS_VIEW, &matTranspose);
```

```
// Создаем матрицу перемещения сетки
D3DXMatrixTranslation(&matTrans, XCoord, YCoord, ZCoord);

// Перемножаем матрицы, чтобы получить матрицу мирового преобразования
D3DXMatrixMultiply(&matWorld, &matTranspose, &matTrans);

// Устанавливаем матрицу мирового преобразования
g_pD3DDevice->SetTransform(D3DTS_WORLD, &matWorld);

// Далее можно рисовать буфер вершин, который будет выровнен
// относительно точки просмотра, с правильными координатами
```

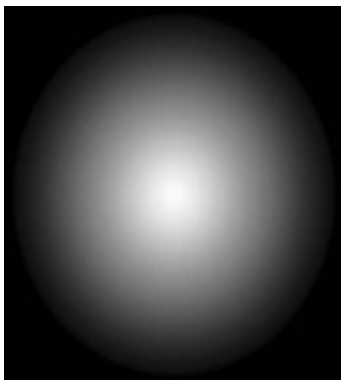
Щиты — это мощная техника, которая является основой для ряда других эффектов, таких как частицы.

## Частицы

Огромные взрывы, дымовые следы и даже крошечные искры, образующие след, тянущийся за магической ракетой, — все это результат работы спецэффекта, известного как *частицы* (*particles*). Частицы руководствуются теми же принципами, что и щиты, и достаточно просты в использовании. Для частицы вы устанавливаете полигон, на который накладывается текстура с изображением дыма, огня, искры, или другого, требуемого вам объекта. В соответствующее время вы разрешаете альфа-смешивание (необязательно) и рисуете частицу, чтобы она была обращена к зрителю (используя щиты). В результате вы получаете набор смешиваемых объектов, которые можно применять для некоторых сногшибательных эффектов.

Одна из замечательных особенностей частиц — то, что они могут быть практически любого размера, поскольку вы можете создать матрицу масштабирования и комбинировать ее с матрицей мирового преобразования полигона частицы. Это означает, что вам надо использовать единственный полигон для рисования всех ваших частиц, кроме тех случаев, когда для частиц используются различные текстуры, тогда количество полигонов должно соответствовать количеству текстур.

Настало время создать изображение частицы. Вы могли бы начать с круга, который является непрозрачным в центре и постепенно, по мере приближения к краю, становится все более прозрачным (как показано на рис. 2.22).



**Рис. 2.22.** Обычно для частиц вы будете использовать изображение круга, как показано здесь. Когда вы используете материалы, изображение при рисовании окрашивается

Теперь устанавливаем четыре вершины, образующие два полигона (в целях оптимизации мы используем полосу треугольников). Координаты вершин определяют исходный размер частицы, который в дальнейшем вы сможете масштабировать. У каждой частицы есть собственные уникальные свойства, включая ее собственный цвет (который реализуется с помощью материала).

Затем вы используете эту структуру в комбинации с единым буфером вершин, содержащим два полигона (образующие квадрат), чтобы визуализировать полигоны на устройстве трехмерной графики. Перед началом рисования каждая частица ориентируется посредством собственной мировой матрицы (конечно же, здесь используются щиты). Вы комбинируете матрицу мирового преобразования частицы с ее же матрицей преобразования масштабирования. Затем вы устанавливаете материал (используя функцию **IDirect3DDevice9::SetMaterial**) для изменения цвета частицы и, наконец, рисуете частицу.

Вот пример, который создает буфер вершин частицы и рисует его на видеоустройстве:

```
// g_pD3DDevice = ранее инициализированный объект устройства
// Определяем структуру данных вершины и дескриптор
typedef struct {
    FLOAT x, y, z; // Локальные 3-D координаты
    FLOAT u, v;    // Координаты текстуры
} sVertex;
#define VertexFVF (D3DFVF_XYZ | D3DFVF_TEX1)

// Буфер вершин частицы и текстура
IDirect3DVertexBuffer9 *g_pParticleVB = NULL;
IDirect3DTexture9 *g_pParticleTexture = NULL;

BOOL SetupParticle()
{
    BYTE *Ptr;
    sVertex Verts[4] = {
        { -1.0f, 1.0f, 0.0f, 0.0f, 0.0f },
        { 1.0f, 1.0f, 0.0f, 1.0f, 0.0f },
        { -1.0f, -1.0f, 0.0f, 0.0f, 1.0f },
        { 1.0f, -1.0f, 0.0f, 1.0f, 1.0f }
    };

    // Создаем буфер вершин частицы и заполняем его данными
    if(FAILED(g_pD3DDevice->CreateVertexBuffer(
        sizeof(sVertex) * 4, 0, VertexFVF,
        D3DPOOL_MANAGED, &g_pParticleVB, NULL)))
        return FALSE;

    if(FAILED(g_pParticleVB->Lock(0,0, (void**)&Ptr, 0)))
        return FALSE;

    memcpy(Ptr, Verts, sizeof(Verts));
    g_pParticleVB->Unlock();

    // Получаем текстуру частицы
    D3DXCreateTextureFromFile(g_pD3DDevice, "particle.bmp",
        &g_pParticleTexture);
}
```



```
        return TRUE;
    }

    BOOL DrawParticle(float x, float y, float z, float scale)
    {
        D3DXMATRIX matWorld, matView, matTransposed;
        D3DXMATRIX matTrans, matScale;
        D3DMATERIAL9 d3dm;

        // Устанавливаем состояния визуализации
        // (альфа-смешивание и атрибуты)
        g_pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
        g_pD3DDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
        g_pD3DDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);

        // Включаем фоновый свет
        g_pD3DDevice->SetRenderState(D3DRS_AMBIENT, 0xffffffff);

        // Устанавливаем буфер вершин частицы в качестве
        // источника потоковых данных
        g_pD3DDevice->SetStreamSource(0, g_pParticleVB,
                                     0, sizeof(sVertex));

        // Настраиваем вершинный шейдер на тип вершин частицы
        g_pD3DDevice->SetFVF(VertexFVF);

        // Устанавливаем текстуру
        g_pD3DDevice->SetTexture(0, g_pParticleTexture);

        // Устанавливаем цвет частицы
        ZeroMemory(&d3dm, sizeof(D3DMATERIAL9));
        d3dm.Diffuse.r = d3dm.Ambient.r = 1.0f;
        d3dm.Diffuse.g = d3dm.Ambient.g = 1.0f;
        d3dm.Diffuse.b = d3dm.Ambient.b = 0.0f;
        d3dm.Diffuse.a = d3dm.Ambient.a = 1.0f;
        g_pD3DDevice->SetMaterial(&d3dm);

        // Строим матрицу масштабирования
        D3DXMatrixScaling(&matScale, scale, scale, scale);

        // Строим матрицу перемещения
        D3DXMatrixTranslation(&matTrans, x, y, z);

        // Строим матрицу вида
        g_pD3DDevice->GetTransform(D3DTS_VIEW, &matView);
        D3DXMatrixTranspose(&matTransposed, &matView);

        // Комбинируем матрицы в единую матрицу мирового преобразования
        D3DXMatrixMultiply(&matWorld, &matScale, &matTransposed);
        D3DXMatrixMultiply(&matWorld, &matWorld, &matTrans);

        // Устанавливаем мировое преобразование
        g_pD3DDevice->SetTransform(D3DTS_WORLD, &matWorld);

        // Рисуем частицу
        g_pD3DDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);

        // Выключаем альфа-смешивание
        g_pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, FALSE);

        return TRUE;
    }
}
```

Эти две функции демонстрируют установку буфера вершин и текстуры, используемых для частицы, и рисование самой частицы. Код полного примера работы с частицами довольно длинный. Я же хотел вам только дать основные понятия о том, как работать с единственной частицей.

Законченный пример приложения, демонстрирующего достаточно сложный вариант применения частиц, вы найдете в проекте Particle на сопроводительном CD-ROM к этой книге (загляните в папку \BookCode\Chap02\Particle).

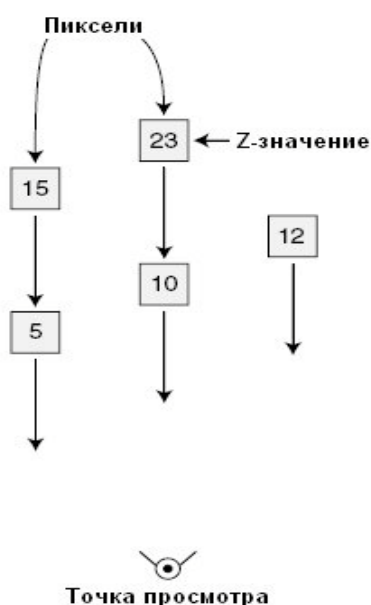
## Сортировка по глубине и Z-буферизация

Легко понять, что когда вы рисуете составляющие сцену объекты сеток, те объекты, которые находятся далеко от зрителя, должны закрываться теми объектами, которые расположены ближе к зрителю. Это называется *сортировкой по глубине (depth sorting)* и для нее есть два общепринятых метода.

Первый метод называется *алгоритм художника (painter's algorithm)*. В нем объекты разделяются на образующие их полигоны, которые затем сортируются в порядке от дальних к ближним, чтобы затем рисовать их в таком порядке. Рисование подобным образом гарантирует, что полигон всегда будет рисоваться поверх тех полигонов, которые находятся за ним.

Второй, и наиболее часто используемый в видеокартах, метод сортировки по глубине называется *методом Z-буфера (Z-Buffer method)*. Он работает попиксельно, назначая каждому пикселю z-значение (расстояние от зрителя).

При записи каждого пикселя визуализатор проверяет, нет ли уже на том же самом месте пикселя с меньшим z-значением. Если нет, пиксель рисуется; если есть — пиксель игнорируется. Иллюстрация этой концепции приведена на рис. 2.23.



**Рис. 2.23.** Когда один пиксель перекрывается другим, будет нарисован только пиксель с меньшим z-значением. Здесь будут нарисованы только три пикселя, поскольку два более далеких пикселя закрываются близлежащими пикселями

У большинства современных видеокарт есть встроенный Z-буфер, поэтому предпочтительнее использовать этот метод сортировки по глубине. Простейший способ использовать Z-буфер в вашем приложении заключается в его инициализации при создании объекта устройства и установке параметров показа.

Сперва нужно выбрать точность буфера (16, 24 или 32 разряда), указав соответствующее значение **D3DFORMAT**. Вы можете обнаружить, что вам на выбор предлагается несколько значений для Z-буфера, но я сосредоточусь на использовании **D3DFMT\_D16** (16-разрядный) и **D3DFMT\_D32** (32-разрядный).

Различная точность используется по двум причинам — из-за занимаемой памяти и из-за качества. В плане занимаемой памяти 32-разрядный Z-буфер требует гораздо больше места для хранения, чем 16-разрядный, так что везде, где возможно, старайтесь обходиться 16-разрядным Z-буфером.

Что касается качества, то при наличии множества перекрывающихся объектов использование 16-разрядного буфера может приводить к тому, что из-за малой точности буфера будет нарисован неправильный пиксель. Переход к 32-разрядному буферу решает вопросы точности, но за счет того, что Z-буфер будет использовать в два раза больше памяти. Вам не стоит беспокоиться; в мире компьютерных игр скорость и оптимизация более важны, так что оставайтесь с 16-разрядным Z-буфером.

Вернемся назад, к установке параметров показа, где для разрешения использования Z-буфера в вашем приложении надо добавить следующие две строки кода:

```
d3dp.EnableAutoDepthStencil = TRUE;  
d3dp.AutoDepthStencilFormat = D3DFMT_D16; // или D3DFMT_D32
```

Теперь вы можете продолжить вашу процедуру инициализации. Когда вы будете готовы к визуализации с использованием Z-буфера (это не определяется автоматически), необходимо установить соответствующие режимы визуализации:

```
// g_pD3DDevice = ранее инициализированный объект устройства  
  
// Для включения Z-буфера используйте:  
g_pD3DDevice->SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);  
  
// Для выключения Z-буфера используйте:  
g_pD3DDevice->SetRenderState(D3DRS_ZENABLE, D3DZB_FALSE);
```

---

**ПРИМЕЧАНИЕ**

Хотя может казаться логичным держать Z-буфер включенным все время, можно получить некоторый выигрыш в скорости, если отключать его, когда он не нужен. Когда вы рисуете изображения, такие как экранное меню, вам нужен полный контроль над тем что и где будет нарисовано, так что не стесняйтесь включать и выключать Z-буфер как считаете целесообразным.

---

Если вы уже попробовали новые возможности Z-буфера, то, возможно, заметили, что кое-что работает неправильно. Например, после нескольких кадров экран не обновляется, или расположенные вдалеке объекты обрезаются. Это вызвано тем, что вы должны очищать Z-буфер перед каждым кадром, а также необходимо скорректировать матрицу проекции, чтобы учесть заданные параметры расстояния.

Чтобы очищать Z-буфер перед каждым кадром, замените вызов **IDirect3DDevice9::Clear** на следующий:

```
g_pD3DDevice->Clear(0, NULL,
                    D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                    D3DCOLOR_RGBA(0.0f, 0.0f, 0.0f, 0.0f), 1.0f, 0);
```

Обратите внимание на появившийся в коде флаг **D3DCLEAR\_ZBUFFER**. Он сообщает функции очистки, что надо очистить Z-буфер, заполнив его указанным значением (пятый аргумент, который в данном примере равен 1.0). Значение должно находиться в диапазоне от 0.0 (минимальная глубина Z-буфера) до 1.0 (максимальная глубина Z-буфера).

Использование значения 1.0 говорит функции очистки, что все значения глубины нужно установить на максимум. Если вы хотите при очистке Z-буфера установить половину максимального значения, укажите в параметре 0.5. Остался только один вопрос — что такое максимальное значение?

Z-буфер измеряет расстояние от точки просмотра, и при визуализации объекты, находящиеся дальше максимальной дистанции просмотра (и те объекты, которые находятся слишком близко) не рисуются. Минимальная и максимальная дистанция просмотра задаются при инициализации матрицы проекции (с помощью D3DX), как показано в следующем фрагменте кода:

```
// Устанавливаем минимальную дистанцию 1.0, а максимальную 1000.0
D3DXMatrixPerspectiveFovLH(&MatrixProj, D3DX_PI/4,
                           1.0f, 1.0f, 1000.0f);
```

Последние два значения, это именно то, что нам надо корректировать — минимальная и максимальная дистанция соответственно. Поэкспериментируйте с этими значениями, чтобы увидеть, как они работают; следует помнить, что увеличение расстояния между минимальной и максимальной дистанциями ухудшает качество Z-буферизации. Обычные значения дистанций: 1.0 для минимума и 1000.0, 2000.0 или 5000.0 для максимума.

## Работа с портом просмотра

Время от времени вам может потребоваться выполнять визуализацию в небольшие области экрана, что-то вроде визуализации в небольшое окно, расположенное внутри главного окна приложения. Главный порт просмотра обычно охватывает весь экран, но когда необходимо, вы можете изменить размер порта просмотра, чтобы он охватывал небольшую часть экрана

(например, для визуализации экрана радара, зеркала заднего вида или других ситуаций «картинка в картинке»).

Чтобы установить порт просмотра вы сначала заполняете структуру **D3DVIEWPORT9**, содержащую координаты и размеры нового порта просмотра, который вы намереваетесь использовать:

```
typedef struct _D3DVIEWPORT9 {
    DWORD X;           // Левая координата X порта просмотра
    DWORD Y;           // Верхняя координата Y порта просмотра
    DWORD Width;       // Ширина порта просмотра
    DWORD Height;      // Высота порта просмотра
    float MinZ;        // 0.0
    float MaxZ;        // 1.0
} D3DVIEWPORT9;
```

После того, как структура заполнена необходимыми данными, вы сообщаете Direct3D, что надо использовать новый порт просмотра, вызвав функцию **ID3DDevice9::SetViewport**, как показано в следующем фрагменте кода:

```
// pD3DDevice = ранее инициализированный объект устройства

// Создаем порт просмотра
D3DVIEWPORT9 d3dvp = { 0,0, 100, 100, 0.0f, 1.0f };

// Устанавливаем новый порт просмотра
pD3DDevice->SetViewport(&d3dvp);
```

С этого момента (после вызова **SetViewport**) вся графика будет визуализироваться в определенное вами окно порта просмотра. После того, как закончите работать с новым портом просмотра, восстановите старый. Чтобы получить параметры старого порта просмотра для их последующего восстановления вызовите функцию **ID3DDevice9::GetViewport**, как показано в следующем коде:

```
// pD3DDevice = ранее инициализированный объект устройства
D3DVIEWPORT9 OldViewport;

// Получаем параметры старого порта просмотра
pD3DDevice->GetViewport (&OldViewport);

// .. меняем параметры порта просмотра, как необходимо

// Восстанавливаем старый порт просмотра
pD3DDevice->SetViewport (&OldViewport);
```

## Работа с сетками

На нижнем уровне Direct3D не работает с сетками, а только с полигонами. Библиотека D3DX расширяет функциональность системы Direct3D, предоставляя вам ряд объектов, которые поддерживают хранение и визуализацию сеток.

На низшем уровне сетки состоят из тысяч вершин и полигонов, что требует сложных манипуляций. К счастью, Direct3D предоставляет собственный формат файлов трехмерных моделей, предназначенный для хранения описывающей трехмерную модель информации, включая (но не ограничиваясь) вершины, грани, нормали и данные текстур. Этот формат файлов известен как .X.

## Х-файлы

Х-файлы (известные своим расширением .X) — это универсальный формат хранения трехмерных моделей, являющийся собственностью Microsoft. (Извините, но мы не зайдем с визитом к агентам Малдеру и Скалли.) Это управляемый шаблонами и полностью расширяемый формат, а значит, вы можете использовать его для всех потребностей хранения данных. В нашем случае под хранением данных я подразумеваю хранение информации ваших трехмерных сеток.

Хотя я мог бы сейчас порассуждать о запутанности формата .X, это перегрузит главу информацией, которой смогут воспользоваться только профессиональные программисты и компьютерные художники. Взгляните правее в лицо — попытка вручную редактировать сетку, состоящую из тысяч вершин, смехотворна. Да и зачем решать эту приводящую в уныние задачу, когда можно воспользоваться такими программами, как trueSpace или MilkShape 3D и разрабатывать сетки в дружелюбной среде? Так я и подумал!

---

### ПРИМЕЧАНИЕ

Если вас действительно заинтересовала запутанность формата Х-файлов, вы, определенно, должны взглянуть на мою книгу «Профессиональная анимация с DirectX». В ней я детально рассматриваю использование Х-файлов в ваших собственных проектах, от применения стандартных объектов данных DirectX до создания ваших собственных настраиваемых наборов данных! Дополнительную информацию о книге вы найдете в приложении А, «Список литературы».

---

Вместо этого, позвольте мне предоставить вам краткий обзор форматирования Х-файлов; после этого вы сможете перейти к действительно интересным вещам — использованию моделей в ваших играх!

## Формат Х-файлов

Х-файлы, идентифицируемые по расширению файла .X, очень гибкие. Они могут быть текстовыми, что упрощает редактирование файла, или двоичными, что уменьшает размер файла и упрощает его защиту от любопытных глаз. В целом формат .X базируется на шаблонах, которые похожи на структуры C.

Чтобы прочитать и обработать Х-файл вы применяете небольшой набор СОМ-объектов, которые разбирают каждый объект данных, встречающийся в Х-файле. Объекты данных обрабатываются как массивы байтов; вы просто

выполняете приведение типа массива к годной для использования структуре, чтобы получить легкий доступ к содержащимся в объекте данным.

Эти объекты могут сильно изменяться, в зависимости от того, что хранится в X-файле. В нашем случае объекты представляют сетки и относящиеся к сеткам данные (такие, как структура костей, называемая иерархией фреймов, и данные анимации). Ваша задача — разобрать эти объекты, загрузить данные сеток, построить таблицы анимации и создать иерархии фреймов.

Я еще вернусь к сеткам и вопросам анимации. Сперва я хочу поговорить о такой вещи первостепенной важности, как упомянутая выше иерархия фреймов.

### **Использование иерархии фреймов**

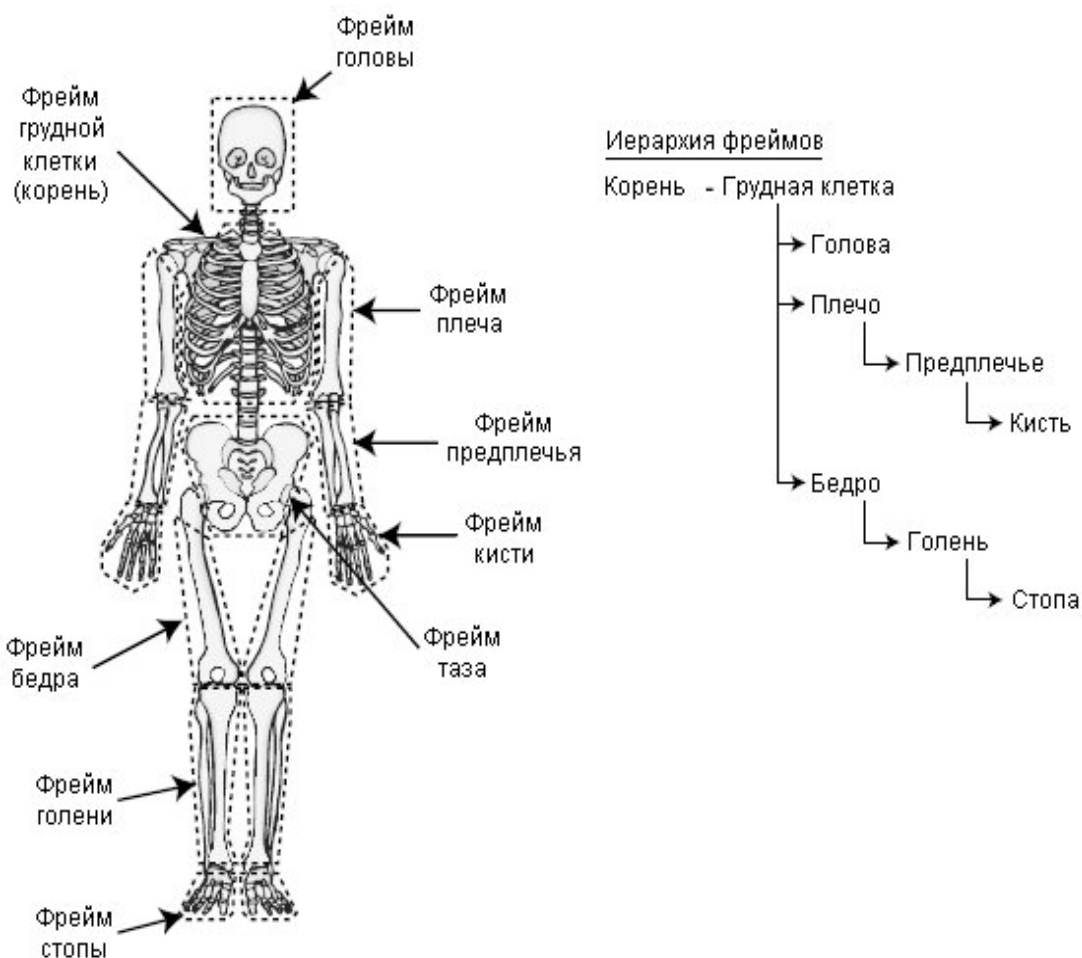
Вы будете использовать *фреймы* (*frame*) для группировки одного или нескольких объектов данных (обычно сеток), чтобы упростить управление ими. Вы можете также создать единственную сетку и использовать несколько фреймов, содержащих ссылку на сетку, что позволяет вам использовать одну сетку несколько раз.

Предположим, у вас есть сетка, изображающая бильярдный шар. Поскольку в наборе 15 шаров, вы создаете 15 фреймов, каждый из которых содержит ссылку на оригинальную сетку шара. После этого вы можете ориентировать каждый фрейм (используя матрицу преобразования фрейма) относительно бильярдного стола, заставляя каждый экземпляр сетки перемещаться вместе с его фреймом.

По сути, вы создали 15 экземпляров сетки бильярдного шара из единственной сетки. Помимо использования фреймов для создания нескольких экземпляров единственной сетки, вы также можете применять их для организации иерархии фреймов. *Иерархия фреймов* (*frame hierarchy*) определяет структуру сцены или группировку сеток. Когда перемещается фрейм, перемещаются также и все встроены в него фреймы.

Для примера представим как набор фреймов скелет человека (рис. 2.24). Вершиной иерархии является грудная клетка. Идя вниз от грудной клетки, вы подсоединяете каждую кость к предыдущей — то есть таз к грудной клетке, бедро к тазу, голень к бедру и стопу к голени. В обратном направлении (вверх) к грудной клетке подсоединяется рука, а к руке — кисть. Это упорядочивание продолжается пока не будут подсоединены все кости так, чтобы на конце цепочки всегда была грудная клетка.

Итак, у вас есть *корневой фрейм* (*root frame*) — грудная клетка. У корня нет родительского фрейма, значит он находится на верху иерархии и не принадлежит другому фрейму. Фреймы, которые подсоединены к другим фреймам, называются *дочерними фреймами* (*child frame*) или, иногда, *узлами* (*node*). Например, плечо является родителем предплечья, а кисть является потомком предплечья. Если следовать иерархии, кисть, предплечье и плечо являются потомками грудной клетки.



**Рис. 2.24.** Ваш скелет является замечательным примером иерархии фреймов. Каждая кость в иерархии возвращается к грудной клетке

Когда перемещается фрейм, перемещаются и все его потомки. Если вы двигаете плечом, предплечье и кисть тоже перемещаются. С другой стороны (здесь нет игры слов), если вы двигаете кистью, положение меняет только она, потому что у нее нет дочерних фреймов (предплечье является родительским фреймом кисти).

У каждого фрейма есть собственная ориентация, которая в терминах X-файла называется *преобразованием фрейма (frame transformation)*. Вы применяете преобразование к самому верхнему объекту в иерархии, для которого оно нужно. Преобразование распространяется от вершины иерархии вниз ко всем дочерним фреймам. Например, если вы поворачиваете плечо, преобразование вращения распространяется вниз к предплечью и кисти (и у каждого фрейма комбинируется с его собственным преобразованием).

Иерархия фреймов является основой для использования сложных сеток и анимационных техник. Фактически, она абсолютно необходима для использования таких вещей, как скелетные сетки (подробнее о скелетных сетках мы поговорим в разделе «Сетки в библиотеке D3DX» данной главы).

Другой причиной для использования иерархии фреймов является изоляция частей сцены. Благодаря ей вы можете модифицировать небольшой



фрагмент сцены, перемещая указанные фреймы, в то время как остальная сцена остается неизменной. Например, если у вас есть фрейм, представляющий дом, и другой фрейм, представляющий дверь, то вы можете изменять фрейм двери не изменяя фрейм дома.

## Создание X-файлов с сетками

Вы можете создавать собственные X-файлы различными способами. Microsoft написала несколько программ экспорта, которые вы можете использовать с программами моделирования, такими как 3D Studio Max или Maya. Если вы не можете приобрести эти замечательные программы моделирования, можно создавать модели вручную (самостоятельно добавляя данные каждой вершины и полигона в текстовый X-файл) или воспользоваться более дешевыми программами, такими как низкополигональный редактор моделей MilkShape 3D, разработанный Митом Кайрэганом.

Программа MilkShape 3D разрабатывалась как инструментальное средство создания моделей для игры Half-Life, но превратилась в нечто гораздо большее. Сейчас MilkShape 3D поддерживает множество форматов моделей (в основном для игр), но остается полезной программой.

Теперь кое-что проясняется и я покажу вам кота в мешке — в MilkShape 3D есть программа экспорта в X-файлы, написанная вашим любимым автором (правильно, мной!), которую вы можете использовать. Перейдите на официальный сайт MilkShape 3D (<http://www.swiss-quake.ch/chumbalum-soft>) чтобы загрузить его.

Если вы не хотите использовать для создания X-файлов программы сторонних производителей, единственным вариантом остается создание моделей вручную. Однако, поскольку самостоятельное создание моделей не является оптимальным решением, в этой книге оно не рассматривается.

## Разбор X-файлов

Позднее в этой главе и далее во всей книге вам надо будет вручную выполнять разбор X-файлов, чтобы получить данные сеток. Для разбора X-файлов используется семейство объектов **IDirectXFile**, которые реализуют задачу открытия X-файла и перечисления содержащихся в файле объектов данных, предоставляя таким образом для вас легкий доступ к ним.

---

**ПРИМЕЧАНИЕ**

Чтобы использовать в вашем проекте компоненты IDirectXFile, вам необходимо включить в проект заголовочные файлы `dxfile.h`, `rmxfguid.h` и `rmxftmpl.h`, а также указать при компоновке библиотеки `dxguid.lib` и `d3dxof.lib`.

---

Разбор X-файлов не так труден, как может показаться с первого взгляда. Трок заключается в просмотре всей иерархии объектов данных и поиске тех объектов, которые вы хотите использовать — обычно это объекты сеток и

фреймов. Главное помнить, что объекты могут быть вложенными и вместо объекта вы можете обнаружить ссылку на него (для которой необходимо выполнить разрешение, чтобы получить доступ к данным объекта). Организация объектов данных показана на рис. 2.25.



*Рис. 2.25. X-файлы позволяют вам встраивать одни объекты в другие, создавая иерархию*

Приведенный ниже код открывает X-файл и выполняет разбор содержащихся в нем объектов. Помните, что эти функции важны для последующего использования в этой главе (и в оставшейся части книги), так что сейчас не следует слишком беспокоиться о том, как применять ее.

```

BOOL ParseXFile(char *Filename)
{
    IDirectXFile          *pDXFile = NULL;
    IDirectXFileEnumObject *pDXEnum = NULL;
    IDirectXFileData      *pDXData = NULL;

    // Создание объекта X-файла
    if(FAILED(DirectXFileCreate(&pDXFile)))
        return FALSE;

    // Регистрация используемых шаблонов
    // Используем стандартные шаблоны
    // абстрактного режима из Direct3D
    if(FAILED(pDXFile->RegisterTemplates((LPVOID)
        D3DRM_XTEMPLATES, D3DRM_XTEMPLATE_BYTES))) {
        pDXFile->Release();
        return FALSE;
    }

    // Создаем объект перечисления
    if(FAILED(pDXFile->CreateEnumObject((LPVOID)Filename,
        DXFILELOAD_FROMFILE, &pDXEnum))) {
        pDXFile->Release();
        return FALSE;
    }

    // Перечисляем все объекты верхнего уровня
    while(SUCCEEDED(pDXEnum->GetNextDataObject(&pDXData))) {
        ParseXFileData(pDXData);
        ReleaseCOM(pDXData);
    }

    // Освобождаем объекты
    ReleaseCOM(pDXEnum);
    ReleaseCOM(pDXFile);

    // Возвращаем флаг успеха
    return TRUE;
}

```

```
void ParseXFileData(IDirectXFileData *pData)
{
    IDirectXFileObject      *pSubObj  = NULL;
    IDirectXFileData        *pSubData = NULL;
    IDirectXFileDataReference *pDataRef = NULL;
    const GUID *pType = NULL;
    char      *pName = NULL;
    DWORD     dwSize;
    char      *pBuffer;

    // Получаем тип объекта
    if(FAILED(pData->GetType(&pType)))
        return;

    // Получаем имя объекта (если есть)
    if(FAILED(pData->GetName(NULL, &dwSize)))
        return;

    if(dwSize) {
        if((pName = new char[dwSize]) != NULL)
            pData->GetName(pName, &dwSize);
    }

    // Если имени нет, задаем имя по умолчанию
    if(pName == NULL) {
        if((pName = new char[9]) == NULL)
            return;
        strcpy(pName, "Template");
    }

    // Смотрим чем является объект
    // Здесь вы перейдете к вашему собственному коду
    // Сканируем встроенные объекты
    while(SUCCEEDED(pData->GetNextObject(&pSubObj))) {
        // Обрабатываем встроенные ссылки
        if(SUCCEEDED(pSubObj->QueryInterface(
            IID_IDirectXFileDataReference,
            (void**)&pDataRef))) {
            if(SUCCEEDED(pDataRef->Resolve(&pSubData))) {
                ParseXFileData(pSubData);
                ReleaseCOM(pSubData);
            }
            ReleaseCOM(pDataRef);
        }
        // Обрабатываем встроенные объекты,
        // не являющиеся ссылками
        if(SUCCEEDED(pSubObj->QueryInterface(
            IID_IDirectXFileData, (void**)&pSubData))) {
            ParseXFileData(pSubData);
            ReleaseCOM(pSubData);
        }
        ReleaseCOM(pSubObj);
    }
    // Освобождаем буфер имени
    delete[] pName;
}
```

Функции **ParseXFile** и **ParseXFileData** работают сообща, чтобы выполнить разбор каждого объекта, содержащегося в X-файле. Функция **ParseXFile** открывает X-файл и выполняет перечисление, определяя

верхние объекты иерархии. Каждый обнаруженный объект передается функции **ParseXFileData**.

Функция **ParseXFileData** обрабатывает данные объекта. Она начинается с получения типа объекта и имени экземпляра объекта (если оно есть). Теперь вы можете обработать данные объекта и позволить функции перечислить все дочерние объекты, используя рекурсию. Этот процесс продолжается, пока не будут обработаны все объекты данных.

Вы просто вызываете функцию **ParseXFile**, передавая ей имя файла, который хотите обработать, а эти две функции позаботятся обо всем остальном. Вы узнаете как использовать эти функции в разделе «Скелетные сетки», далее в этой главе.

## Сетки в библиотеке D3DX

Чаще всего вы будете иметь дело с двумя типами сеток Direct3D: *стандартными сетками* (*standard mesh*) и *скелетными сетками* (*skinned mesh*). Стандартные сетки, они и есть стандартные. У них нет никаких прикрас, за исключением возможности использовать текстуры для улучшения внешнего вида.

Скелетные сетки уникальны тем, что они *деформируемые* (*deformable*). Это значит, что сетка может динамически изменять свою форму во время работы программы. Чтобы подготовить сетку к деформации вы должны в вашей программе моделирования присоединить вершины сетки к воображаемому набору костей. При перемещении костей будут перемещаться и присоединенные к ним вершины.

Перед тем, как более подробно поговорить о стандартных и скелетных сетках, давайте взглянем на особый объект, используемый обоими типами сеток для хранения данных — **ID3DXBuffer**.

### Объект ID3DXBuffer

Объект **ID3DXBuffer** используется для хранения и восстановления буферов данных. Библиотека D3DX применяет объект **ID3DXBuffer** для хранения информации о сетках, такой как материалы и списки текстур. О том как действует объект буфера данных вы подробнее узнаете в разделе «Стандартные сетки» этой главы.

У интерфейса **ID3DXBuffer** есть всего две функции. Первая — это **ID3DXBuffer::GetBufferPointer**, которая применяется для получения указателя на данные, хранящиеся в буфере объекта. Вызов функции **GetBufferPointer** возвращает указатель типа **void**, который вы можете привести к любому типу данных.

```
void *ID3DXBuffer::GetBufferPointer();
```

Вторая функция — это **ID3DXBuffer::GetBufferSize**, которая возвращает количество байтов, выделенных для хранения данных.

```
DWORD ID3DXBuffer::GetBufferSize();
```

Вы можете создать объект **ID3DXBuffer** для своих собственных нужд с помощью функции **D3DXCreateBuffer**:

```
HRESULT D3DXCreateBuffer(  
    DWORD          NumBytes,      // Размер создаваемого буфера  
    ID3DXBuffer **ppvBuffer); // Созданный объект буфера
```

Что хорошего в прототипе функции без примера ее использования — так вот он (создание объекта буфера размером 1024 байта и заполнение его нулями):

```
ID3DXBuffer *pBuffer;  
  
// Создаем буфер  
if(SUCCEEDED(D3DXCreateBuffer(1024, &pBuffer))) {  
  
    // Получаем указатель на буфер  
    char *pPtr = pBuffer->GetBufferPoint();  
  
    // Заполняем буфер нулями  
    memset(pPtr, 0, pBuffer->GetBufferSize());  
  
    // Освобождаем буфер  
    pBuffer->Release();  
}
```

## Стандартные сетки

Стандартные сетки очень просты; они содержат только описание сетки. С этими сетками проще всего работать, так что с них лучше всего начинать обучение. Еще более упрощает работу со стандартными сетками использование библиотеки D3DX, поскольку для загрузки и отображения стандартной сетки в D3DX требуется всего несколько строк кода. Стандартные сетки, с которыми я буду работать в книге, представляются объектом **ID3DXMesh**, который отвечает за хранение и рисование единственной сетки.

После объявления экземпляра объекта **ID3DXMesh**, воспользуйтесь следующей функцией для загрузки объекта с сеткой из X-файла:

```
HRESULT D3DXLoadMeshFromX(  
    LPSTR          pFilename,      // Имя X-файла  
    DWORD          Options,        // D3DXMESH_SYSTEMMEM  
    IDirect3DDevice9 *pDevice,     // Инициализированный объект  
                                // устройства  
    ID3DXBuffer **ppAdjacency,    // NULL  
    ID3DXBuffer **ppMaterials,    // Буфер для данных материалов  
    DWORD          pNumMaterials,  // Количество материалов в сетке  
    ID3DXMesh      **ppMesh);     // Создаваемый объект сетки
```

Большинство аргументов функции **D3DXLoadMeshFromX** заполняются библиотекой D3DX во время работы функции. Вы указываете имя загружаемого X-файла, неинициализированные объекты **ID3DXBuffer** и **ID3DXMesh** и переменную типа **DWORD** для хранения количества используемых в сетке материалов.

Если вы попытаетесь загрузить X-файл, содержащий несколько сеток, функция **D3DXLoadMeshFromX** объединит их все в единую сетку. На данном этапе это нас вполне устраивает. Взгляните на фрагмент рабочего кода, который загружает единственную сетку:

```
// g_pD3DDevice = ранее инициализированный объект устройства
ID3DXBuffer *pD3DXMaterials;
DWORD      g_dwNumMaterials;
ID3DXMesh  *g_pD3DXMesh;

if(FAILED(D3DXLoadMeshFromX("mesh.x", D3DXMESH_SYSTEMMEM,
                           g_pD3DDevice, NULL, &pD3DXMaterials,
                           &g_dwNumMaterials, &g_pD3DXMesh))) {
    // Произошла ошибка
}
```

После успешной загрузки вашей сетки вы запрашиваете информацию о материалах и текстурах, используя следующий фрагмент кода:

```
D3DXMATERIAL      *pMaterials      = NULL;
D3DMATERIAL9      *g_pMaterialList = NULL;
IDirect3DTexture9 **g_pTextureList;

// Получаем указатель на список материалов
pMaterials = (D3DXMATERIAL*)pD3DXMaterials->GetBufferPointer();

if(pMaterials != NULL) {
    // Создаем массив структур данных материалов
    // для копирования в него данных
    g_pMaterialList = new D3DMATERIAL9[dwNumMaterials];

    // Создаем массив указателей на объекты текстуры
    g_pTextureList = new IDirect3DTexture9[dwNumMaterials];

    // Копируем материалы
    for(DWORD i = 0; i < dwNumMaterials; i++) {
        g_pMaterialList[i] = pMaterials[i].MatD3D;

        // Делаем фоновую составляющую цвета такой же,
        // как и рассеиваемая
        g_pMaterialList[i].Ambient = g_pMaterialList[i].Diffuse;

        // Создаем и загружаем текстуры (если они есть)
        if(FAILED(D3DXCreateTextureFromFileA(g_pD3DDevice,
            g_pMaterials[i]->pTextureFilename, &g_pTextureList[i])))
            g_pTextureList[i] = NULL;
    }

    // Освобождаем буфер материалов, использовавшийся для загрузки
    pD3DXMaterials->Release();
} else {
```

```
// Если материалы не были загружены, создаем
// материал по умолчанию
g_dwNumMaterials = 1;

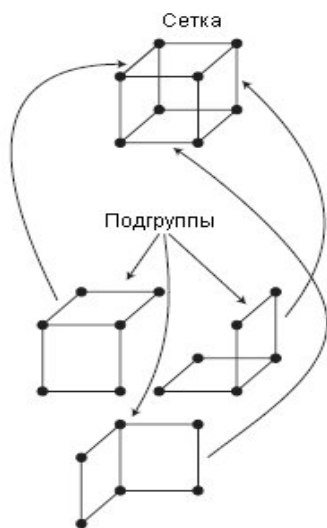
// Создаем белый материал
g_pMaterialList = new D3DMATERIAL9[1];
g_pMaterialList[i].Diffuse.r = 1.0f;
g_pMaterialList[i].Diffuse.g = 1.0f;
g_pMaterialList[i].Diffuse.b = 1.0f;
g_pMaterialList[i].Diffuse.a = 1.0f;
g_pMaterialList[i].Ambient = g_pMaterialList[i].Diffuse;

// Создаем пустую ссылку на текстуру
g_pTextureList = new IDirect3DTexture9[1];
g_pTextureList[0] = NULL;
}
```

После завершения показанного выше фрагмента кода, вы получаете замечательный новый список материалов и текстур, все элементы которого инициализированы и готовы к использованию в ваших сценах. Единственная оставшаяся задача — визуализация вашей сетки.

## Визуализация сетки

Сердцем объекта **ID3DXMesh** является единственная функция визуализации с именем **DrawSubset**, которая выполняет работу по визуализации подгруппы сетки. *Подгруппа (subset)* — это часть сетки отделенная по причине смены параметров визуализации по сравнению с предыдущей подгруппой, например из-за смены материала или текстуры. Вы можете разделить сетку на несколько подгрупп (например, как показано на рис. 2.26). Ваша задача — понять, что представляет каждая подгруппа и визуализировать ее.



**Рис. 2.26.** Подгруппы используются для разделения отдельных частей сетки

После загрузки X-файла у вас будет объект сетки и материалы. Подгруппы сетки связаны с этими материалами, так что если у вас в сетке есть пять материалов, это значит, что сетка содержит пять подгрупп для рисования.

Такое упорядочивание подгрупп облегчает визуализацию сетки; просто переберите в цикле материалы, установите каждый материал и визуализируйте соответствующую подгруппу. Повторяйте эти действия, пока не будет нарисована вся сетка. Чтобы разместить сетку в мире установите перед ее рисованием матрицу мирового преобразования. Вот пример, использующий ранее загруженную сетку:

```
// g_pD3DDevice = ранее инициализированный объект устройства
// pD3DXMesh     = ранее загруженный объект ID3DXMesh
// matWorld       = матрица мирового преобразования сетки

// Начало сцены
if(SUCCEEDED(g_pD3DDevice->BeginScene())) {
    // Устанавливаем матрицу мирового преобразования сетки
    g_pD3DDevice->SetTransform(D3DTS_WORLD, &matWorld);

    // Перебираем в цикле каждый материал в сетке
    for(DWORD i = 0; i < g_dwNumMaterials; i++) {
        // Устанавливаем материал и текстуру
        g_pD3DDevice->SetMaterial(&g_pMaterialList[i]);
        g_pD3DDevice->SetTexture(0, g_pTextureList[i]);

        // Рисуем подгруппу сетки
        pD3DXMesh->DrawSubset(i);
    }
    // Завершение сцены
    g_pD3DDevice->EndScene();
}
```

Помните, что прежде чем визуализировать сетку, вам надо установить ее матрицу мирового преобразования, чтобы поместить ее где-нибудь под каким-нибудь углом в вашем трехмерном мире. Если вы загрузили несколько сеток, то можете соединить их в форме объекта, анимируемого путем изменения ориентации отдельных сеток. Это основы трехмерной анимации (подробнее об этой теме мы поговорим в разделе «X-стиль трехмерной анимации», далее в этой главе).

## Скелетные сетки

Одна из самых захватывающих возможностей Direct3D — скелетные сетки. Как я уже упоминал, скелетные сетки могут динамически деформироваться. Это достигается путем подсоединения отдельных вершин сетки к структуре лежащих в основе «костей» или иерархии фреймов. Скелетные сетки используют кости для определения своей формы; при перемещении костей сетка соответствующим образом деформируется.

Кости представляются как иерархия фреймов внутри X-файла. При моделировании сетки вы выстраиваете фреймы в структуру «предок–потомок». Когда изменяется ориентация родительского фрейма, присоединенные к нему дочерние фреймы наследуют родительское преобразование и комбинируют его со своими собственными преобразованиями. Это упрощает анимацию — вы перемещаете



единственный фрейм и все, присоединенные к нему фреймы перемещаются следом.

Для загрузки и использования скелетных сеток вы имеете дело непосредственно с объектами данных X-файла, как это делалось ранее в разделе «Разбор X-файлов». (Я же говорил вам, что этот код еще пригодится.) Для разбора объектов вам потребуется управлять списком фреймов (и их иерархией).

### **Загрузка скелетных сеток**

В процессе перечисления объектов данных, содержащихся в X-файле, вам потребуется вызывать различные функции загрузки сеток библиотеки D3DX для обработки объектов данных. Одна из представляющих для нас интерес функций, которую мы будем применять для загрузки скелетных сеток, — это **D3DXLoadSkinMeshFromXof**. Она выполняет работу по чтению объекта данных сетки из X-файла, созданию содержащего сетку объекта **ID3DXMesh** и созданию объекта **ID3DXSkinInfo**, описывающего соединение костей и вершин, что необходимо для деформации сетки.

Поскольку код для разбора иерархии фреймов и загрузки сеток достаточно сложен, он приводится на сопроводительном CD-ROM к этой книге (в папке \BookCode\Chap02\XFile). Я слышу ваши вздохи, но не волнуйтесь — код хорошо прокомментирован и я прямо сейчас уделю время его исследованию. Сперва вы находите несколько структур, содержащих иерархию фреймов (заполненную матрицами преобразования фреймов) и сетки.

Функция **LoadMesh** использует слегка модифицированную версию показанной ранее (в разделе «Разбор X-файлов») функции разбора. Объекты данных фреймов, после того, как они перечислены в функции **LoadFile**, добавляются к иерархии фреймов. Затем перечисляются другие объекты, находящиеся внутри объектов фреймов, обеспечивая иерархию наборами дочерних объектов.

Если в процессе перечисления объектов из X-файла будет обнаружен объект данных сетки, функция **LoadFile** загрузит сетку с помощью функции **D3DXLoadSkinMeshFromXof**. Затем загруженный объект сетки добавляется к связанному списку сеток. Фреймы содержат указатели на сетки, так что вы можете использовать одну и ту же сетку несколько раз (используя обращение по ссылке).

После того, как сетка загружена, функция загрузки сопоставляет кости сетки с соответствующими им фреймами и загружает материалы сетки.

Чтобы загрузить скелетную сетку или набор сеток, вызовите функцию **LoadMesh**, передав ей имя X-файла. В свою очередь, **LoadFile** вызывает функцию **ParseXFile**. Затем, после возврата из функции **LoadFile**, вы получаете указатель на фрейм, который является корнем для всех других фреймов.

Чтобы визуализировать скелетную сетку, вы должны сперва ее обновить. Обновление сетки берет вершины загруженной сетки, применяет к ним различные преобразования (из иерархии костей) и сохраняет преобразованные вершины во второй контейнер сетки, который вы будете использовать для визуализации скелетной сетки.

Абсолютно верно, для использования скелетной сетки вам необходимо создать два контейнера сеток. Первая сетка — это та, которая загружена из X-файла. Это всего лишь стандартная сетка, использующая объект **ID3DXMesh** для хранения своих данных. Ваша задача сейчас — клонировать (дублировать) эту сетку, чтобы использовать дубликат при визуализации деформированной скелетной сетки.

Для создания клонированной сетки, вызовите функцию объекта сетки **CloneMeshFVF**:

```
HRESULT ID3DXMesh::CloneMeshFVF(
    DWORD          Options,          // Параметры сетки (установите 0)
    DWORD          FVF,              // Новый FVF
    LPDIRECT3DDevice9 pDevice,        // Устройство для создания сетки
    LPD3DXMESH      *ppCloneMesh); // Новый объект сетки
```

Поскольку вы создаете точный дубликат оригинальной сетки, можно использовать следующий фрагмент кода, делающий это для вас:

```
// pMesh = объект ID3DXMesh, загруженный из X-файла
ID3DXMesh *pSkinMesh = NULL;
pMesh->CloneMeshFVF(0, pMesh->GetFvF(), pD3DDevice, &pSkinMesh);
```

Теперь, когда у вас есть второй контейнер сетки (объект скелетной сетки), вы можете начать изменять структуру костей скелетной сетки, обновлять секту и визуализировать результат на экране!

## **Обновление и визуализация скелетной сетки**

Чтобы модифицировать преобразования костей, вы создаете массив объектов **D3DXMATRIX** (по одной матрице для кости). Так, если в вашей скелетной сетке используется 10 костей, то вам необходимо 10 объектов **D3DXMATRIX** для хранения преобразований каждой кости.

Пойдемте дальше и сохраним различные преобразования для каждой кости. Очень важно помнить, что каждая кость наследует преобразования своих предков, поэтому при изменении ориентации кости проявляется эффект распространяющейся вниз ряби.

Когда вы закончите играть с преобразованиями, можно обновить вершины скелетной сетки и визуализировать сетку. Для обновления скелетной сетки вам необходимо заблокировать буфер вершин исходной сетки и буфер вершин скелетной сетки (созданного вами клона сетки):

```
void *Src, *Dest;
pMesh->LockVertexBuffer(0, (void**)&Src);
pSkinnedMesh->LockVertexBuffer(0, (void**)&Dest);
```

После блокировки вызовите функцию **ID3DXSkinInfo::UpdateSkinnedMesh**, чтобы преобразовать все вершины скелетной сетки согласно ориентации костей, заданной в массиве объектов **D3DXMATRIX**. В приведенном ниже фрагменте подразумевается, что массив объектов **D3DXMATRIX** называется **matTransforms**:

```
// matTransforms = массив объектов D3DXMATRIX
// pSkinInfo      = объект D3DXSkinInfo, полученный при
//                вызове D3DXLoadSkinMeshFromXof.
pSkinInfo->UpdateSkinnedMesh(matTransforms, NULL, Src, Dest);
```

Теперь просто разблокируйте буферы вершин и визуализируйте контейнер скелетной сетки (клонированную сетку):

```
pSkinnedMesh->UnlockVertexBuffer();
pMesh->UnlockVertexBuffer();

// Визуализируем объект pSkinnedMesh, используя ту же технику
// что и ранее в этой главе. (Помните, что pSkinnedMesh
// это просто объект ID3DXMesh - используйте материалы и
// текстуры pMesh перед вызовом pSkinnedMesh->DrawSubset.
```

Я знаю, что сказал очень мало, и материал, возможно, трудно понять, но дело в том, что тема скелетных сеток слишком обширна, чтобы ее можно было рассмотреть на нескольких страницах. Для лучшего понимания того, как работать со скелетными сетками в собственных проектах, я настоятельно рекомендую вам изучить демонстрационные примеры, находящиеся на сопроводительном CD-ROM.

В конце этой главы вы найдете краткую информацию о демонстрационном примере XFile, который захотите изучить, поскольку он показывает как загрузить и визуализировать скелетную сетку.

## Х-стиль трехмерной анимации

Трехмерная анимация полностью отличается от двухмерной. Вам больше не предоставляется удовольствие рисовать множество изображений, а затем последовательно показывать их для создания анимации. В трехмерном мире на объект можно посмотреть практически под любым углом.

Основа трехмерной анимации — изменение матриц преобразования фреймов, используемых для ориентации ваших сеток во время выполнения, что приводит к изменению местоположения расположенных во фреймах сеток. Это перемещение сеток и есть анимация. Вы можете передвигать, вращать и даже масштабировать сетки как захотите.

Когда вы имеете дело со скелетными сетками, использование преобразований фреймов — это единственный способ анимации сетки. Поскольку скелетная сетка — это единая сетка (она не составлена из нескольких сеток), для того, чтобы деформировать вершины вам надо менять фреймы. Простейший способ модифицировать матрицы преобразования

фреймов — это воспользоваться техникой, называемой *ключевые кадры* (*key framing*).

## Техника ключевых кадров

В компьютерной анимации *ключевыми кадрами* (*key frame*) называется техника, в которой берется две завершённые ориентации объекта (ключевые кадры) и выполняется интерполяция между ними на основании какого-нибудь фактора, например, времени. Другими словами, зная ориентацию объекта в двух различных кадрах (каждый со своей ориентацией, называемой *ключом*), вы можете вычислить его ориентацию в любой момент времени между этими ключами (как показано на рис. 2.27).



**Рис. 2.27.** Ориентация фреймов интерполируется в зависимости от прошедшего с начала времени

Техника ключевых кадров эффективно использует память и гарантирует, что анимация будет выполняться с одинаковой скоростью на всех системах. Более медленные компьютеры пропускают кадры (за счет плавности анимации), а более быстрые компьютеры генерируют больше кадров, обеспечивая более плавную анимацию.

Как вы уже знаете, иерархия фреймов создается из подсоединенных один к другому фреймов. Вы также знаете, как анимировать фреймы сетки используя объекты анимации X-файла. Теперь вам только надо выполнить вычисления интерполяции между этими фреймами, чтобы получить плавную анимацию.

Тот способ использования ключевых кадров, который я показываю в этой книге, — это *матрицы ключевых кадров* (*matrix key framing*). Поскольку вы уже используете объекты матриц D3DX, данную форму ключевых кадров использовать легко. Предположим, у вас есть две матрицы: **mat1** и **mat2**, являющиеся начальной и конечной матрицей соответственно. Промежуток времени между ними представлен как **Length**, а текущее время

представлено как **Time** (в диапазоне от 0 до **Length**). Вы вычисляете интерполированную матрицу следующим образом:

```
// D3DXMATRIX Mat1, Mat2;  
// DWORD Length, Time;  
D3DXMATRIX MatInt; // Итоговая интерполированная матрица  
  
// Вычисляем интерполированную матрицу  
MatInt = (Mat2 - Mat1) / Length;  
  
// Умножаем на время  
MatInt *= Time;  
  
// Добавляем Mat1 и это результат!  
MatInt += Mat1;
```

---

**ПРИМЕЧАНИЕ**

*Интерполяция (interpolating)* — это способ вычисления промежуточных значений между двумя числами на основании времени. Например, если ваш дом находится в двух милях от работы, и путь до работы занимает у вас 30 минут, вы можете определить расстояние до работы в любой момент времени, используя следующее вычисление интерполяции:

$$\text{Distance} = (\text{DistanceToWork} / \text{TravelTime}) * \text{CurrentTime};$$

Как видите, через 26 минут вы проедете  $((2 / 30) * 26) = 1.73$  мили.

---

Заключительные вычисления завершены, и вы остаетесь с матрицей, содержащей ориентацию где-нибудь между используемыми матрицами.

## Анимация в X-файлах

Microsoft обеспечила хранение данных анимации в X-файлах. Данные анимации находятся внутри ряда специальных объектов данных, и вы можете загружать данные из этих объектов анимации используя ту же самую технику, которую применяли для загрузки скелетных сеток.

Загрузка анимации из X-файла очень беспорядочна; есть анимация в целом, каждая с объектом анимации, объектом анимационного набора, объектом времени, объектами ключевых кадров — с какой кучей вещей приходится иметь дело!

Вместо того, чтобы здесь пробираться через создание пакета анимации, обратитесь к коду на прилагаемом к книге CD-ROM (загляните в папку \BookCode\Chap02\XFile). Вы можете также обратиться к главе 6, «Создание ядра игры», чтобы исследовать законченный пакет анимации, созданный для этой книги. Пока, тем не менее, продолжите чтение этого раздела, чтобы узнать как работает анимация в X-файлах.

Специальные объекты содержат различные ключи, применяемые в технике ключевых кадров. Каждый ключ представляет единственное преобразование: вращение, масштабирование и перемещение. Для простоты вы можете задать матрицу ключа в которой все преобразования

скомбинированы воедино (именно такой тип ключей я буду использовать в этой книге).

У каждого ключа есть связанное с ним время, в которое он должен стать активным. Другими словами, ключ вращения с  $\text{time} = 0$  означает, что когда время равно 0 используются значения вращения из ключа. Второй ключ вращения становится активным когда  $\text{time} = 200$ . Пока время идет, вычисляются интерполированные значения вращения, находящиеся где-нибудь между первым и вторым ключами вращения. Когда время достигает 200, значение вращения равно тому, которое указано во втором ключе. Такая форма интерполяции применяется к значениям ключей всех типов.

---

**ПРИМЕЧАНИЕ**

Поймите, что когда мы здесь обсуждаем время, это не какое-то реальное измерение. Вы сами должны решить, как измерять его. Например, время может быть количеством секунд, прошедших после какого-то момента, или время может быть количеством сформированных кадров. Для простоты время должно быть основано на системном времени компьютера, что позволяет синхронизировать анимацию с точностью до секунды (или, впрочем, до миллисекунды).

---

Анимации входят в наборы и каждому набору назначен определенный фрейм. Вы можете назначить одному и тому же набору несколько ключей, чтобы несколько ключей влияло на фрейм. Например фрейм может модифицироваться набором ключей вращения и набором ключей перемещения одновременно, так что фрейм будет перемещаться и вращаться одновременно.

И снова на помощь приходят программы трехмерного моделирования, такие как 3D Studio Max или MilkShape 3D; благодаря им вам никогда не придется иметь дело непосредственно с данными анимации в X-файле. Кроме того, используя приведенный в этой книге код, вы можете в ваших играх загружать и показывать полностью анимированные сетки, абсолютно не тратя времени! Более полную информацию о загрузке и использовании анимации вы найдете в главе 6.

## Заканчиваем с графикой

Поздравляю! Завершился стремительный тур по DirectX Graphics! Вы так много увидели и узнали, что хорошо бы уделить некоторое время и убедиться, что вы полностью поняли основы — вершины, полигоны, текстуры и материалы. Понимание этих основ — единственный билет к сногсшибательным эффектам показанным в этой главе и далее в книге.

## Программы на CD-ROM

Программы, демонстрирующие обсуждавшийся в этой главе код, находятся на прилагаемом к книге CD-ROM. В папке \BookCode\Chap02\ вы найдете следующие программы:

**Draw2D** — программа показывает как рисовать полигоны (равномерно закрашенные и текстурированные) используя двухмерные координаты. Местоположение: \BookCode\Chap02\Draw2D\.

**Draw3D** — программа показывает как рисовать полигоны (равномерно закрашенные и текстурированные) используя трехмерные координаты и трехмерные преобразования. Местоположение: \BookCode\Chap02\Draw3D\.

**Alpha** — программа демонстрирует использование альфа-смешивания и копирования с учетом прозрачности. Местоположение: \BookCode\Chap02\Alpha\.

**Lights** — с этой программой вы увидите эффект, оказываемый на объекты источниками света разных типов. Местоположение: \BookCode\Chap02\Lights\.

**ZBuffer** — программа демонстрирует использование Z-буфера для сортировки по глубине. Местоположение: \BookCode\Chap02\ZBuffer\.

**Font** — программа показывает, как использовать возможности работы со шрифтами, обсуждавшиеся в этой главе. Местоположение: \BookCode\Chap02\Font\.

**Particle** — программа рисует набор перемещающихся частиц, используя технику щитов. Местоположение: \BookCode\Chap02\Particle\.

**XFile** — программа демонстрирует использование функций работы с сетками библиотеки D3DX. Местоположение: \BookCode\Chap02\XFile\.

# Глава 3

## Взаимодействие с пользователем с DirectInput

Одна вещь определенно справедлива для большинства приложений — они нуждаются в пользовательском вводе. Используете ли вы джойстик, чтобы вести вашего героя по экрану, щелкаете ли кнопкой мыши по ссылке на Web-странице или просто печатаете письмо с помощью клавиатуры — результаты всех рассмотренных действий должны быть введены. Разве не было бы замечательно использовать в вашей игре все эти три устройства ввода? И, раз существует множество различных устройств, не можем ли мы поддерживать их все?

Не беспокойтесь; Microsoft приходит на помощь с DirectInput! Не надо больше беспокоиться о мельчайших деталях каждого устройства ввода. DirectInput предоставит вам простой, обобщенный метод доступа к ним.

Эта глава начинается с краткого обсуждения принципов работы устройств ввода и затем перемещается на их использование с DirectInput. Если более конкретно, в этой главе вы узнаете следующее:

- Использование устройств ввода.
- Работа с DirectInput.
- Инициализация DirectInput.
- Использование устройств DirectInput.
- Чтение и использование данных устройств.

### Знакомство с устройствами ввода

Наиболее часто в компьютерах используется три устройства ввода: клавиатура, мышь и джойстик. Все они неоценимы, но у каждого есть свои сильные и слабые стороны. Клавиатура хороша, когда надо что-то напечатать, но не заменит мышь и слишком велика, чтобы использовать ее как джойстик.

Что касается мыши, она замечательно подходит для указания и перемещения, но ей недостает простоты управления джойстика. Джойстик великолепен для простых перемещений — вверх, вниз, влево и вправо — но



почти бесполезен для чего-нибудь другого. Как говорится, «Ни с ним, ни без него».

Каждое устройство ввода взаимодействует с компьютером своим собственным способом — некоторые через драйвера устройств (небольшие программы, единственная цель которых — иметь дело с устройствами ввода), тогда как другие имеют дело непосредственно с памятью компьютера. Ваша задача — запросить у драйвера устройства или операционной системы любую информацию, которую может передавать устройство ввода, и затем обработать эту информацию, как считаете нужным.

## Взаимодействие через клавиатуру

Нажмите клавишу на клавиатуре и соответствующая буква, цифра или символ магическим образом появятся на экране. Однако, взаимодействие между клавиатурой и компьютером не так просто, и вы, как программист игр, должны в нем разбираться.

Фактически, клавиатура — это набор логически упорядоченных клавиш. За исключением нескольких различий, у всех клавиатур есть стандартная раскладка. Каждая клавиша представляет собой переключатель, который может быть либо замкнут (клавиша нажата), либо разомкнут (клавиша отпущена).

Нажатие или отпускание клавиши посылает сигнал микропроцессору клавиатуры, который, в свою очередь, генерирует сигнал для компьютера, называемый *прерывание* (*interrupt*). В свою очередь система получает данные от микропроцессора клавиатуры, позволяющие установить, какая клавиша была нажата или отпущена. Эти данные называются *скан-кодами* (*scan code*).

---

**ПРИМЕЧАНИЕ** *Прерывание* (*interrupt*) — это сигнал, сообщаящий системе, что устройство или программа нуждаются в обслуживании как можно скорее. Использование прерываний гарантирует, что система будет знать об изменении состояния устройства.

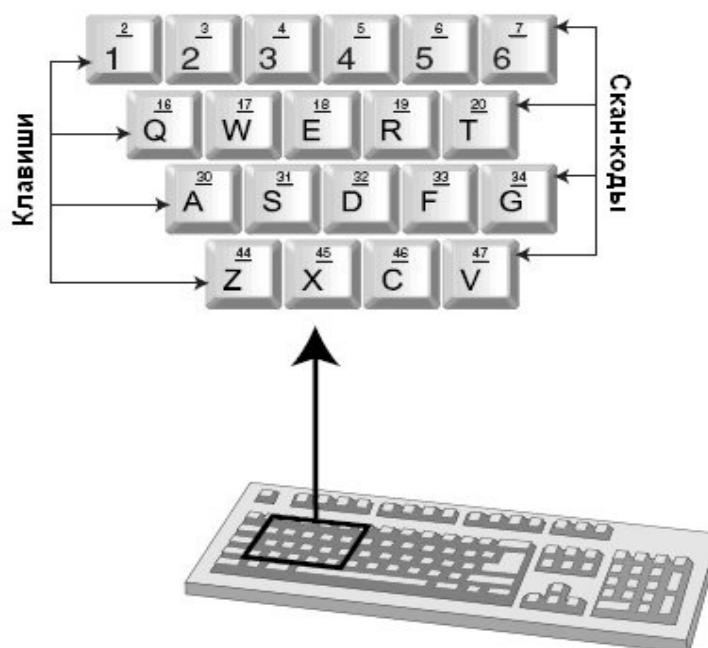
*Скан-коды* (*scan code*) — это значения, определяющие, какая клавиша была нажата или отпущена. Эти коды представляют собой единственный байт (хотя вам система может передавать их в различном виде), в котором часть разрядов определяют клавишу, а один бит сообщает, была нажата эта клавиша или отпущена.

---

Поскольку различные клавиатуры могут иметь различную раскладку, скан-коды могут изменяться в зависимости от клавиатуры. Однако, это не является проблемой, поскольку большинство клавиатур поддерживает стандартную 101 – 102 клавишную раскладку. Если вы не поняли, у каждой клавиатуры есть раскладка, ссылающаяся на клавиши по их номеру, и в большинстве случаев этих клавиш 101 или 102.

**ПРИМЕЧАНИЕ** Поскольку скан-коды различают нажатие только одной клавиши, в них нет разницы между заглавными и строчными буквами. Так, скан-код для заглавной **A** и для строчной **a** будет одним и тем же. Различия между заглавными и строчными буквами определяет операционная система, отслеживая состояние клавиши **Shift**.

На рис. 3.1 показана часть раскладки скан-кодов для обычной 101 – 102 клавишной клавиатуры. Обратите внимание, что скан-коды упорядочены согласно расположению клавиш на клавиатуре; в ряду клавиш та клавиша, которая расположена правее обычно имеет большее значение скан-кода.



**Рис. 3.1.** Фрагмент клавиатуры, показывающий несколько клавиш с их скан-кодами. Как видите, скан-коды это просто порядковые номера

Младшие 7 бит скан-кода (биты с 0 по 6) определяют клавишу (какая клавиша была нажата или отпущена), а старший бит скан кода (бит 7) указывает была ли эта клавиша нажата (бит установлен) или отпущена (бит сброшен). Наибольшее число, которое может быть представлено байтом, равно 255, а старший бит зарезервирован, так что значений остается для 128 клавиш.

### **Работа с клавиатурой в Windows**

Windows может выполнять задачу ввода с клавиатуры за вас. Чтобы облегчить вашу задачу Windows конвертирует поступающие от клавиатуры скан-коды (зависящие от клавиатуры или чего-нибудь еще) в стандартизованные значения, называемые *виртуальными кодами клавиш* (*virtual key codes*) и *ASCII-кодами* (*ASCII codes*). Windows каким-либо способом передает эти коды программисту, обычно через процедуру обработки сообщений.

**ПРИМЕЧАНИЕ**

Виртуальные коды клавиш (Virtual Key Code) — это Windows-версия скан-кодов. Вместо того, чтобы использовать для буквы **A** скан-код 30 (и надеяться, что это правильно), вы применяете макрос виртуального кода клавиши для буквы A, **VK\_A**, и будете уверены, что он всегда соответствует букве A, независимо от того, какая клавиатура используется (и скан-кода, сообщаемого клавиатурой).

ASCII (*Американский Стандартный Код для Обмена Информацией*, *American Standart Code for Information Interchange*) — это стандарт, диктующий какое значение какому символу должно соответствовать. С ASCII вы можете различать заглавные и строчные буквы, поскольку им соответствуют различные значения. ASCII устанавливает значения для 128 различных символов, среди которых есть цифры, буквы, общеупотребительные знаки и управляющие символы.

Windows обычно использует коды *Расширенного ASCII* (*Extended ASCII*) и коды *Unicode* (или *широкие символы*, *wide character*). Расширенный ASCII добавляет еще один бит к значениям обычного ASCII, что увеличивает количество представляемых символов до 256. Недостаток этого варианта заключается в отсутствии стандарта. Кроме того, некоторым языкам требуется больше символов, и поэтому был разработан Unicode, в котором для хранения символов используется 16 бит, что достаточно для 65 536 символов.

---

Что вы будете делать с полученными виртуальными кодами клавиш или кодами ASCII зависит от вас и вашего приложения. В текстовом редакторе вы можете вставить соответствующую клавише букву в тело текста. В игре вы можете обработать нажатие клавиши и переместить персонаж игрока на экране.

Я только что сказал про использование клавиатуры в играх? Да, верно — вы можете весьма эффективно использовать клавиатуру для игры. Единственная проблема в том, что Windows может не поспевать за бешеным темпом нажатий на клавиши в некоторых играх. Каково же решение? Вы узнаете о нем чуть позже, в разделе «Использование DirectInput».

## Играем с мышью

Если несколько друзей — друзья, значит несколько мышей — мышья? Ладно, оставим эту тему для споров другим; меня вполне устраивает вариант «мыши». В отличие от своих пушистых тезок, компьютерные мыши для нас очень полезны.

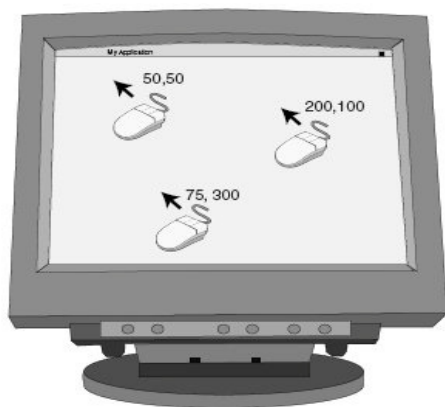
Мышь состоит всего из нескольких компонентов. Обычно это две или три кнопки и шарик снизу, отслеживающий перемещения мыши. С кнопками все просто — нажмите ее, и компьютеру будет отправлен сигнал; отпустите,

и компьютер снова получит сигнал. Некоторые мыши для отслеживания перемещений используют оптический сенсор и поэтому не нуждаются в шарике, но делают они абсолютно то же самое.

В действительности под твердой спинкой маленького грызуна нет ничего сложного для понимания. На самом нижнем уровне мышь просто сообщает системе, что она передвигается в определенном направлении — по одному сигналу за раз. Драйвер читает эти данные и преобразует сигналы в соответствующие значения перемещения.

В обычном приложении Windows берет эти перемещения и сообщает о них в сообщениях пользователю через процедуру обработки сообщений. Работа процедуры обработки сообщений весьма медленна — каждое сообщение, передаваемое этой процедуре, помещается в очередь, обрабатывающую события в порядке их добавления. Чтобы ускорить процесс получения и обработки данных мыши, вы обращаетесь напрямую к драйверу мыши, исключая процедуру обработки сообщений Windows из процесса.

Независимо от способа получения информации о перемещении мыши, вы отслеживаете координаты указателя мыши на экране. Можно отслеживать абсолютные или относительные координаты мыши. *Абсолютные* координаты — это текущее местоположение мыши относительно некоторой фиксированной точки (обычно относительно верхнего левого угла экрана, как показано на рис. 3.2). *Относительные* координаты ссылаются на перемещение влево, вправо, вверх или вниз относительно последней зафиксированной позиции. Как вы вскоре увидите, оба способа легко реализуются.



**Рис. 3.2.** Вы измеряете абсолютные координаты мыши в количестве пикселей, на которое указатель мыши отстоит от верхнего левого угла экрана

#### ПРИМЕЧАНИЕ

*Относительное перемещение* — это величина перемещения относительно последней записанной позиции. Например, если вы стоите и пройдете пять шагов вперед, ваша относительная позиция от того места, где вы находились, будет пять шагов назад.

Наименьшее перемещение, о котором может сообщить мышь называется *Микки (Mickey)*. (Какой удар для старины Уолта!) Драйвер мыши отвечает за преобразование этих Микки в значения, которые он может передать системе.

Кнопки мыши могут только сигнализировать о том, нажаты они в данный момент или нет, точно так же, как это делают клавиши клавиатуры.

## Удовольствие с джойстиком

А-ах, ощущение нового джойстика. Отформованные пластиковые детали, резиновая рукоятка и блестящие кнопки — все кажется идеально подходит к вашей руке. Мысли о завоеваниях заполняют голову. Перед вами проходят воспоминания о прошлых победах и поражениях. Кажется, вы рождены для того, чтобы играть.

Джойстики предназначены для управления играми. Хотя это и не единственные устройства ввода, применяемые в играх, они наиболее естественны. Наклоните рукоятку влево, и ваш персонаж переместится влево. Нажмите кнопку, и ваш герой взмахнет мечом. Что может быть проще?

Джойстики бывают самых различных форм и размеров. Руль, который вы видите на полке магазина — это тоже джойстик. Если вы когда-нибудь были в зале игровых автоматов, то наверное играли в игры где для управления персонажем на экране надо стоять на сноуборде или сидеть на мотоцикле. Пусть вас не обманывает внешность — эти сноуборды и мотоциклы также считаются джойстиками!

Джойстик — это устройство в котором есть контроллер оси и несколько кнопок. У руля один контроллер оси для поворотов руля влево и вправо. Могут быть два дополнительных контроллера оси для педалей газа и тормоза. В обычном двухкнопочном джойстике два контроллера оси — один для перемещений вверх и вниз, и другой для перемещений вправо и влево. Варианты джойстиков показаны на рис. 3.3.



**Рис. 3.3.** Задающие направление, поворачивающиеся и нажимаемые приспособления для управления — вот общие черты джойстиков, независимо от их внешнего вида

*Контроллер оси (axis control)* — это просто потенциометр (переменный резистор), регулирующий напряжение в цепи. Наименьшее напряжение в цепи соответствует одному крайнему положению (самой дальней точке, в которую может быть перемещен джойстик) по оси, а наибольшее напряжение — другому крайнему положению. Промежуточные уровни соответствуют положению оси где-нибудь между этими крайними положениями.

Напряжение передается системе, благополучно обрабатывается Windows (или DirectInput) и предоставляется вам для использования. Кнопки джойстика работают почти так же, только наличие или отсутствие напряжения сообщает нажата или отпущена кнопка соответственно.

Данные джойстика представляют собой значения, показывающие отклонение от центральной позиции. Отклоните рукоятку влево или вверх и вы получите отрицательное значение, представляющее расстояние от центра. Отклоните рукоятку вниз или вправо, и вы получите положительные значения. Кнопки представляются отдельными флагами, которые установлены, если кнопка нажата.

---

**ПРИМЕЧАНИЕ**

Единственное большое различие между джойстиками, которое вы можете заметить, заключается в том, что джойстики с цифровыми контроллерами оси работают как набор кнопок. В таких джойстиках отклонение рукоятки влево аналогично нажатию кнопки «влево». Всякий раз, когда программист запрашивает такой джойстик о положении оси, ему будет возвращено минимальное или максимальное значение.

---

## Использование DirectInput

Скорее всего, вы будете использовать клавиатуру и мышь, а так же было бы удобно иметь возможность пользоваться джойстиком. Однако, доступно так много различных типов этих устройств, что голова кружится от одной мысли попытаться поддерживать их все.

Конечно, для общения с мышью и клавиатурой вы можете использовать очередь сообщений Windows, что сделает вашу жизнь легче. Проблема в том, что очередь сообщений медленная — ну сколько можно ждать, пока критически важная информация о нажатии клавиш или перемещении мыши пройдет через очередь сообщений, когда она нужна прямо сейчас?

Встречайте DirectInput — решение ваших проблем. С ним вы получаете метод быстрого получения данных, когда они нужны вам, вместо того, чтобы ждать пока их предоставит Windows. Оказывается, DirectInput самый простой для использования компонент DirectX.

Благодаря DirectInput ваши программы могут легко использовать любые клавиатуры, мыши и джойстики, подключенные к системе пользователя (а также и любые другие устройства ввода, совместимые с DirectInput). А когда вы попытаетесь написать работающий код, то обнаружите, что он занимает всего несколько десятков строк!

## Представляем основы DirectInput

DirectInput представляет собой коллекцию COM-объектов (подобно всем остальным компонентам DirectX) которые представляют систему ввода, а также отдельные устройства ввода (как показано в таблице 3.1). Главный

объект, **IDirectInput8**, используется для инициализации системы и создания интерфейсов устройств ввода.

---

**ПРИМЕЧАНИЕ** Да, вы прочитали правильно, — DirectInput все еще использует объекты версии 8, несмотря на то, что вы установили DirectX SDK версии 9! Это вызвано тем, что по сравнению с 8 версией в DirectInput были сделаны очень незначительные изменения, и Microsoft оставила все как есть. Разве не очаровательно?

---

**Таблица 3.1.** COM-объекты DirectInput

<b>Объект</b>	<b>Описание</b>
<b>IDirectInput8</b>	Главный COM-интерфейс DirectInput 8. Все другие интерфейсы запрашиваются через него.
<b>IDirectInputDevice8</b>	COM-интерфейс для устройств ввода. Каждое устройство имеет отдельный собственный интерфейс для использования.
<b>IDirectInputEffect</b>	COM-интерфейс для эффекта обратной связи, имеющегося в некоторых моделях джойстиков и мышей.

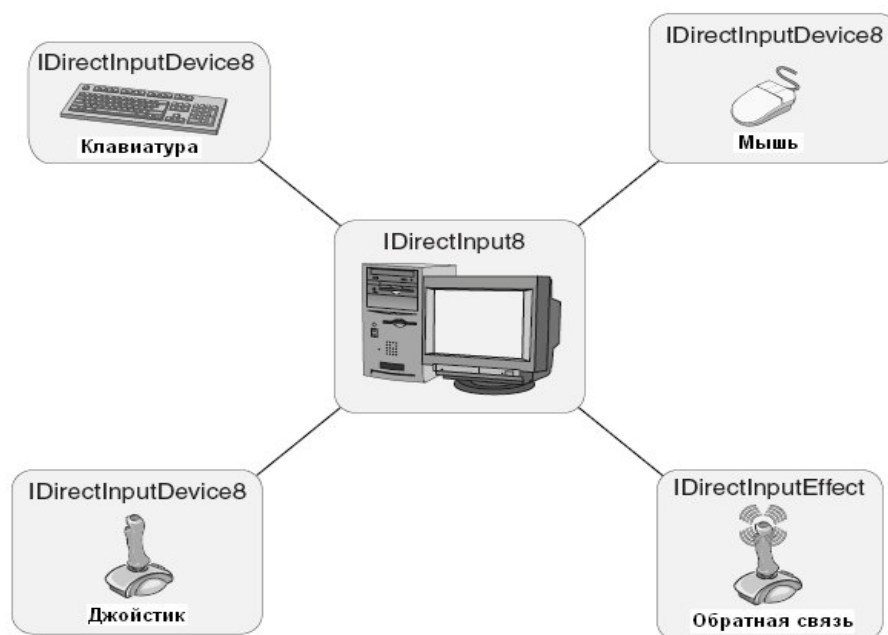
---

**ПРИМЕЧАНИЕ** В этой книге я буду использовать только два первых COM-объекта из таблицы 3.1, а **IDirectInputEffect** останется за кадром.

---

Для работы с каждым устройством ввода (таким, как клавиатура, мышь или джойстик) используется объект общего интерфейса **IDirectInputDevice8**. Некоторые устройства, такие как джойстики и мыши, позволяют отправлять соответствующему им объекту **IDirectInputDevice8** запросы на получение дополнительного интерфейса **IDirectInputEffect**, который применяется для управления поддерживаемыми эффектами обратной связи. Взаимоотношения между интерфейсами **IDirectInput8**, **IDirectInputDevice8** и **IDirectInputEffect** показаны на рис. 3.4.

Объект **IDirectInput8** содержит набор функций, используемых для инициализации системы ввода и получения интерфейсов устройств (которые и будут выполнять всю работу). Из этих функций вам обычно будут нужны только две: **IDirectInput8::EnumDevices** и **IDirectInput8::CreateDevice**. О них вы узнаете больше, когда мы перейдем к непосредственной работе с устройствами ввода в разделе «Используем устройства DirectInput».



*Рис. 3.4. Распорядителем шоу является IDirectInput8, который создает различные объекты IDirectInputDevice8. В свою очередь объекты IDirectInputDevice8 могут использоваться для создания их собственных объектов IDirectInputEffect.*

## Инициализация DirectInput

Перед тем, как начать использовать DirectInput, убедитесь, что в ваш проект включен заголовочный файл `DInput.h` и при компоновке используется библиотека `DInput8.lib`. Главный объект DirectInput представляется объектом **IDirectInput8**, так что мы разместим его в разделе глобальных объявлений:

```
IDirectInput8 g_pDI; // Глобальный объект DirectInput
```

DirectInput предоставляет вспомогательную функцию **DirectInput8Create**, которая инициализирует этот интерфейс для вас. Вот ее прототип:

```
HRESULT WINAPI DirectInput8Create(
    HINSTANCE hInstance, // Дескриптор экземпляра вашей программы
    DWORD dwVersion,     // DIRECTINPUT_VERSION
    REFIID riidltf,      // IID_IDirectInput8
    LPVOID *ppvOut,      // Указатель на новый объект
    LPUNKNOWN pUnkOuter); // Устанавливаем NULL
```

### ПРИМЕЧАНИЕ

Функция **DirectInput8Create**, как и все COM-интерфейсы возвращает значение типа **HRESULT**. Если все завершилось успешно, DirectInput возвращает значение **DI\_OK**, а в случае неудачи — другое значение. Коды ошибок описаны в документации к DirectX SDK. Проверяя возвращаемый код используйте макросы **FAILED** или **SUCCEEDED**. Функция может завершиться удачно, но вернуть код ошибки, а макросы определяют это за вас.



В аргументах нет ничего сложного; вы просто предоставляете указатель на создаваемый объект и устанавливаете остальные параметры, как сказано в комментариях. Приведенный ниже фрагмент кода показывает, как можно создать интерфейс DirectInput:

```
IDirectInput8 *g_pDI; // Глобальный объект DirectInput

int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hPrev,
                  LPSTR szCmdLine, int nCmdShow)
{
    HRESULT hr;
    hr = DirectInput8Create(hInst, DIRECTINPUT_VERSION,
                          IID_IDirectInput8, (void**)&g_pDI, NULL);

    // Сообщаем о сбое, если произошла ошибка
    if(FAILED(hr))
        return FALSE;

    // Переходим к основной программе
```

Вот и все, что относится к инициализации DirectInput! Далее мы узнаем о том, как инициализировать объект устройства для работы с конкретным оборудованием (клавиатурой, мышью и джойстиком).

---

**ПРИМЕЧАНИЕ**

Как и для всех COM-объектов, убедитесь, что вызвали метод **Release**, когда закончили работу.

```
g_pDI->Release();
```

---

## Использование устройств DirectInput

Microsoft проделала длинный путь, чтобы упростить работу с устройствами ввода. Она настолько проста, что фактически вы можете использовать один и тот же COM-интерфейс (**IDirectInputDevice8**) для взаимодействия практически с любым устройством ввода, которое вы себе можете представить (и, возможно, с теми, которые вы и представить не можете!). Шаги для создания и использования различных устройств ввода похожи, так что позвольте мне забежать вперед и показать, как это делается. В таблице 3.2 показаны общие этапы в правильном порядке вместе с интерфейсами DirectInput и функциями, которые данный этап реализуют.

**Таблица 3.2.** Этапы создания и использования устройств

Этап	Интерфейс/функция
1. Получение GUID устройства	<b>IDirectInput8::EnumDevices</b>
2. Создание COM-объекта устройства	<b>IDirectInput8::CreateDevice</b>
3. Установка формата данных	<b>IDirectInputDevice8::SetDataFormat</b>

Таблица 3.2. Этапы создания и использования устройств (продолжение)

Этап	Интерфейс/функция
4. Установка уровня кооперации	<code>IDirectInputDevice8::SetCooperativeLevel</code>
5. Установка различных специальных свойств	<code>IDirectInputDevice8::SetProperty</code>
6. Захват устройства	<code>IDirectInputDevice8::Acquire</code>
7. Опрос устройства	<code>IDirectInputDevice8::Poll</code>
8. Чтение данных	<code>IDirectInputDevice8::GetDeviceState</code>

Перед тем, как двигаться дальше, убедитесь, что вы объявили объект устройства `DirectInput` (**`IDirectInputDevice8`**):

```
IDirectInputDevice8 *pDIDevice;
```

## Получение GUID устройства

У каждого установленного устройства есть GUID (*Глобальный Уникальный Идентификатор*) — присвоенное ему число. Чтобы использовать устройство надо сначала узнать его GUID. Проще всего это сделать для системной клавиатуры и мыши. `DirectInput` определяет их как **`GUID_SysKeyboard`** и **`GUID_SysMouse`** соответственно. Для всех других устройств вы должны сначала выполнить перечисление, чтобы найти то, которое вам требуется.

**ПРИМЕЧАНИЕ** *Перечисление (enumeration)* — это процесс перебора элементов списка. В нашем случае элементы — это устройства ввода, такие как джойстики. Предположим, к вашей системе подключено пять джойстиков. В процессе перечисления `DirectInput` будет передавать вам информацию, относящуюся к каждому джойстику, по одному за раз, пока не будут перебраны все джойстики, или вы не решите остановиться.

**ПРИМЕЧАНИЕ** Чтобы использовать значения **`GUID_SysKeyboard`** и **`GUID_SysMouse`** вы должны определить константу **`INITGUID`** прежде всех других директив препроцессора или включить в проект библиотеку `DXGuid.lib`.

```
#define INITGUID
```

Вы должны определить **`INITGUID`** только один раз в вашем проекте (в одном файле с исходным кодом); если вы сделаете это несколько раз, то при компиляции возникнет ошибка.

Перечисление устройств — это работа функции **`IDirectInput8::EnumDevices`**:

```

HRESULT IDirectInput8::EnumDevices(
    DWORD          dwDevType, // Тип искомого устройств
    LPDIENUMCALLBACK lpCallback, // Функция обратного вызова
                                // для перечисления
    LPVOID          pvRef,      // Указатель на пользовательский
                                // набор данных
    DWORD           dwFlags);    // Флаги перечисления

```

Параметр **dwDevType** — это набор битовых полей, описывающих устройства какого типа будут перечисляться. Возможен диапазон от общих устройств ввода, таких как джойстики и мыши, до более специализированных, таких как трекболы и рукоятки управления. За полным списком устройств обратитесь к документации DX SDK, а в таблице 3.3. приведено несколько наиболее часто используемых значений.

**Таблица 3.3.** Типы перечисляемых устройств DirectInput

<b>Значение</b>	<b>Описание</b>
<b>DI8DEVCLASS_ALL</b>	Все устройства
<b>DI8DEVCLASS_GAMECTRL</b>	Все игровые контроллеры (джойстики)
<b>DI8DEVCLASS_KEYBOARD</b>	Все клавиатуры
<b>DI8DEVCLASS_POINTER</b>	Все указывающие устройства (мыши)
<b>DI8DEVCLASS_DEVICE</b>	Все устройства, не относящиеся к трем предыдущим типам
<b>DI8DEVTYPE_MOUSE</b>	Мыши и подобные им устройства, такие как трекболы
<b>DI8DEVTYPE_KEYBOARD</b>	Клавиатуры и подобные им устройства
<b>DI8DEVTYPE_JOYSTICK</b>	Джойстики и подобные им устройства, такие как рули
<b>DI8DEVTYPE_DEVICE</b>	Устройства, которые не относятся к трем предыдущим типам
<b>DI8DEVTYPEMOUSE_TOUCHPAD</b>	Сенсорные панели (подтип)
<b>DI8DEVTYPEMOUSE_TRACKBALL</b>	Трекболы (подтип)

Переменная **lpCallback** — это указатель на функцию перечисления, которая будет вызываться каждый раз, когда в системе обнаружено подходящее устройство. Я опишу ее чуть позже. В параметре **pvRef** вы задаете указатель на буфер, обычно это структура данных в которой вы храните информацию. При вызове функции перечисления ей будет передан этот указатель на предоставленную вами структуру (или на другие данные), что предоставляет возможность получить или установить данные.

Последний набор флагов, **dwFlags**, сообщает DirectInput как должно выполняться перечисление устройств. Вы можете указать в аргументе **dwFlags** любое значение из перечисленных в таблице 3.4.

Таблица 3.4. Флаги перечисления

Значение	Описание
<b>DIEDFL_ALLDEVICES</b>	Перечисляем все установленные устройства (значение по умолчанию)
<b>DIEDFL_ATTACHEDONLY</b>	Перечисляем только подключенные устройства
<b>DIEDFL_FORCEFEEDBACK</b>	Перечисляем только устройства с обратной связью
<b>DIEDFL_INCLUDEALIASES</b>	Включаем устройства, которые являются псевдонимами других устройств
<b>DIEDFL_INCLUDEPHANTOMS</b>	Включаем фантомные устройства (заглушки)

Ранее упомянутый указатель на функцию **lpCallback** должен указывать на определяемую пользователем функцию перечисления с типом **DIEnumDeviceProc**, которая должна соответствовать следующему прототипу:

```
BOOL CALLBACK DIEnumDevicesProc(
    LPDIDEVICEINSTANCE lpddi, // Структура данных устройства
    LPVOID pvRef); // Задаваемый пользователем указатель
```

Параметр **lpddi** — это указатель на структуру **DIDEVICEINSTANCE**, которая содержит информацию об обнаруженном в данном вызове устройстве. Вот как она выглядит:

```
typedef struct {
    DWORD dwSize; // Размер структуры
    GUID guidInstance; // GUID устройства
    GUID guidProduct; // Предоставляемый OEM GUID устройства
    DWORD dwDevType; // Тип устройства
    TCHAR tszInstanceName[MAX_PATH]; // Название устройства
    TCHAR tszProductName[MAX_PATH]; // Название продукта
    GUID guidFFDriver; // GUID драйвера обратной связи
    WORD wUsagePage; // Используемая страница для устройств HID
    WORD wUsage; // Используемый код для устройств HID
} DIDEVICEINSTANCE;
```

**ПРИМЕЧАНИЕ** При работе со структурой **DIDEVICEINSTANCE** вам потребуются только поля **guidInstance**, **dwDevType** и **tszInstanceName**. Поле **guidInstance** — это GUID, необходимый для инициализации устройства (это именно то, что мы ищем). Поле **dwDevType** — это тип устройства, перечисляемого в данный момент.

И, наконец, **tszInstanceName** — это текстовый буфер, который содержит имя устройства (такое, как «Joystick 1»), которое можно использовать, например, для формирования списка из которого пользователь будет выбирать устройство.

Позвольте мне теперь отнять у вас несколько минут и построить пару функций, которые инициализируют **DirectInput** и перечисляют все

устройства, отображая имена найденных устройств по одному в окне сообщений. При этом вы можете каждый раз выбрать, прекратить перечисление или продолжить, щелкая по кнопке **Cancel** или **OK** соответственно.

---

<b>ПРИМЕЧАНИЕ</b>	Еще одно замечание о перечислении: в функции перечисления вы должны определить надо ли продолжать перечисление, или следует остановиться, для чего необходимо вернуть значение <b>DIENUM_CONTINUE</b> или <b>DIENUM_STOP</b> соответственно.
-------------------	--

---

```
// Глобальный COM-объект DirectInput
IDirectInput8 *g_pDI;

// Прототипы функций
BOOL InitDIAndEnumAllDevices(HWND hWnd, HINSTANCE hInst);
BOOL CALLBACK EnumDevices(LPCDIDEVICEINSTANCE pdInst,
                          LPVOID pvRef);

BOOL InitDIAndEnumAllDevices(HWND hWnd, HINSTANCE hInst)
{
    if(FAILED(DirectInput8Create(hInst, DIRECTINPUT_VERSION,
                                IID_IDirectInput8, (void*)&g_pDI, NULL)))
        return FALSE;

    g_pDI->EnumDevices(DI8DEVCLASS_ALL, EnumDevices,
                      (LPVOID)hWnd, DIEDFL_ALLDEVICES);

    return TRUE;
}

BOOL CALLBACK EnumDevices(LPCDIDEVICEINSTANCE pdInst,
                          LPVOID pvRef)
{
    int Result;

    // Отображаем окно сообщений с именем найденного устройства
    Result = MessageBox((HWND)pvRef, pdInst->tszInstanceName,
                        "Device Found", MB_OKCANCEL);

    // Продолжаем перечисление, если нажата кнопка OK
    if(Result == IDOK)
        return DIENUM_CONTINUE;

    // Завершаем перечисление
    return DIENUM_STOP;
}
```

Просто вставьте приведенный выше фрагмент кода в вашу собственную программу и вызовите функцию **InitDIAndEnumAllDevices()**.

---

<b>ПРИМЕЧАНИЕ</b>	Предположим, вы хотите перечислить только джойстики, подключенные к системе. Просто замените строку <b>g_pDI-&gt;EnumDevices</b> на следующую:
-------------------	--

---

```
g_pDI->EnumDevices(DI8DEVTYPE_JOYSTICK,
                  EnumDevices, (LPVOID)hWnd,
                  DIEDFL_ATTACHEDONLY);
```

---

## Создание COM-объекта устройства

Теперь, когда вы знаете GUID устройства, можно создать COM-объект **IDirectInputDevice8**. Это работа функции **IdirectInput8::CreateDevice**:

```
HRESULT IDirectInput8::CreateDevice(
    REFGUID rguid, // GUID создаваемого устройства жестко
                  // заданный или определенный при перечислении
    LPDIRECTINPUTDEVICE *lplpDirectInputDevice, // Указатель на
                                                  // создаваемый объект
    LPUNKNOWN pUnkOuter); // NULL - не используется
```

Вот пример использования **IDirectInput8::CreateDevice**:

```
IDirectInputDevice8 *pDIDevice;
HRESULT hr = g_pDI->CreateDevice(DeviceGUID, &pDIDevice, NULL);
```

Вот еще пример создания объекта системной клавиатуры с использованием предопределенного GUID:

```
IDirectInputDevice8 *pDIDevice;
HRESULT hr = pDI->CreateDevice(GUID_SysKeyboard, &pDIDevice, NULL);
```

---

**ПРИМЕЧАНИЕ** Вы можете встроить вызов функции **CreateDevice** в функцию перечисления, как будет показано в разделе «DirectInput и джойстики» далее в этой главе.

---

## Установка формата данных

У каждого устройства есть собственный формат данных, используемый при чтении. В нем необходимо учесть множество вещей: клавиши, кнопки, оси и т.д. Чтобы ваша программа могла начать читать данные устройства, необходимо сообщить DirectInput этот формат.

Делается это через функцию **IDirectInputDevice8::SetDataFormat**:

```
HRESULT IDirectInputDevice8::SetDataFormat(LPCDIDATAFORMAT lpdf);
```

У функции **SetDataFormat** только один аргумент, являющийся указателем на структуру **DIDATAFORMAT**, определение которой выглядит так:

```
typedef struct {
    DWORD dwSize; // Размер структуры
    DWORD dwObjSize; // Размер структуры DIOBJECTDATAFORMAT
    DWORD dwFlags; // Флаг, определяющий работает ли устройство
                  // в абсолютном режиме (DIDF_ABSAXIS)
                  // или в относительном режиме (DIDF_RELAXIS)
    DWORD dwDataSize; // Размер получаемого от устройства
                    // пакета данных (в двойных словах)
    DWORD dwNumObjs; // Количество объектов в массиве rgodf
    LPDIOBJECTDATAFORMAT rgodf; // Адрес массива структур
                              // DIOBJECTDATAFORMAT
} DIDATAFORMAT, *LPDIDATAFORMAT;
```

В большинстве случаев вам не придется заниматься инициализацией этой структуры, потому что DirectInput предоставляет вам для использования несколько предопределенных ее экземпляров, которые перечислены в таблице 3.5.

**Таблица 3.5.** Предустановленные структуры данных устройств DirectInput

<i>Устройство</i>	<i>Пример структуры данных</i>
Клавиатура	<code>c_dfDIKeyboard pDIDevice-&gt;SetDataFormat(&amp;c_dfDIKeyboard);</code>
Мышь	<code>c_dfDIMouse pDIDevice-&gt;SetDataFormat(&amp;c_dfDIMouse);</code>
Джойстик	<code>c_dfJoystick pDIDevice-&gt;SetDataFormat(&amp;c_dfDIJoystick);</code>

Я не буду обсуждать специфику создания собственного формата данных, поскольку перечисленных в таблице 3.5 структур данных устройств вполне достаточно для этой книги. Если вы хотите создать собственную структуру формата данных устройства, обратитесь за информацией к документации DirectX SDK.

## Установка уровня кооперации

Взглянем в лицо фактам — каждая программа использует несколько устройств ввода. Почти все программы используют клавиатуру и мышь, а некоторые — еще и джойстик. Когда доходит до дела, вы можете использовать совместный с другими запущенными приложениями доступ к этим устройствам, либо по-хулигански полностью захватить доступ к устройству для своего приложения, не позволяя другим программам управлять им, пока вы не закончите свою работу.

Следующий этап в использовании устройства — установка уровня кооперации, за которую отвечает следующая функция:

```
HRESULT IDirectInputDevice8::SetCooperativeLevel(
    HWND hWnd,          // Дескриптор родительского окна
    DWORD dwFlags);     // Флаги, определяющие вид совместного доступа
```

В аргументе **hWnd** функции **SetCooperativeLevel** укажите дескриптор окна вашего приложения. Чтобы использовать доступ к устройству совместно с другими приложениями, укажите в аргументе **dwFlags** одно из перечисленных в таблице 3.6 значений, которые определяют вид совместного доступа.

Когда вы задаете уровень кооперации, то должны указать либо флаг **DISCL\_EXCLUSIVE**, либо флаг **DISCL\_NONEXCLUSIVE** и скомбинировать его с флагом **DISCL\_BACKGROUND** или **DISCL\_FOREGROUND**. Я рекомендую для рассматриваемых устройств использовать комбинацию **DISCL\_FOREGROUND** и **DISCL\_NONEXCLUSIVE**:

```
pDIDevice->SetCooperativeLevel(hWnd,  
                                DISCL_FOREGROUND | DISCL_NONEXCLUSIVE);
```

**Таблица 3.6.** Предустановленные структуры данных устройств DirectInput

<b>Уровень</b>	<b>Описание</b>
<b>DISCL_NONEXCLUSIVE</b>	Использование данного параметра позволяет нескольким приложениям совместно использовать одно устройство ввода не влияя друг на друга
<b>DISCL_EXCLUSIVE</b>	Этот параметр делает вашу программу захватчиком. При захвате устройства вы получаете единоличный доступ к нему, даже если другие установили эксклюзивный режим использования
<b>DISCL_FOREGROUND</b>	Приложению требуется доступ при активности. Это означает, что для использования устройства программа должна быть активной. Если ваша программа становится неактивной, устройство будет автоматически освобождено, и вам надо будет восстановить контроль, когда программа вновь получит фокус
<b>DISCL_BACKGROUND</b>	Вашей программе требуется фоновый доступ. Это значит, что программа может обращаться к устройству, даже когда она неактивна
<b>DISCL_NOWINKEY</b>	Блокирует клавишу с логотипом Windows
<b>ПРИМЕЧАНИЕ</b>	Использовать флаг <b>DISCL_NOWINKEY</b> необязательно, но я рекомендую указывать его для полноэкранных приложений, чтобы предотвратить нежелательные прерывания работы программы.

## Установка специальных свойств

Теперь вы можете установить любые специальные свойства для устройства, такие как режимы осей, буферизация или максимальные и минимальные значения диапазона. Для режима осей есть два варианта — относительный и абсолютный. Абсолютный режим сообщает координаты относительно центральной точки. Позиции левее или выше этой точки передаются как отрицательные числа, а позиции правее или ниже — как положительные.

Относительные координаты сообщают о перемещении относительно последней зафиксированной позиции. Например, если указатель мыши находился в точке с координатами 100, 40 и переместился на 5 единиц вправо, то DirectInput вернет значение 5, а не 105. Ваша задача — использовать эти относительные перемещения так, как считаете нужным.

Что касается буферизации, то вы можете задать количество сохраняемых в буфере данных (если это необходимо). Это позволяет вам читать поступающие от устройства данные в своем собственном темпе и не беспокоиться о том, что они могут быть потеряны. Использовать



буферизацию данных удобно, но это не то, в чем вы будете нуждаться при разработке игр. Предпочительнее знать состояние устройства ввода в данный момент времени, поэтому я не буду обсуждать буферизацию.

Последние интересные свойства — это минимальное и максимальное значение диапазона для устройства. Например, отклонение джойстика в крайнюю левую позицию приводит к передаче минимального значения, а при отклонении рукоятки в крайнюю правую позицию будет передано максимальное значение. Вы устанавливаете оба эти значения. Они влияют только на джойстики, так что именно их я буду использовать в качестве примера.

Установкой специальных свойств заведует функция **IDirectInputDevice8::SetProperty**:

```
HRESULT IDirectInputDevice8::SetProperty(
    REFGUID rguidProp, // GUID свойства
    LPCDIPROPHEADER pdiph); // DIPROPHEADER содержащий данные
                          // об устанавливаемом свойстве
```

---

**ПРИМЕЧАНИЕ** За один раз может быть установлено только одно свойство. Если необходимо установить другое свойство, вызовите функцию еще раз.

---

GUID, указанный в **rguidProp**, представляет свойство, которое вы хотите установить, такое как диапазон сообщаемых джойстиком значений или режим осей. Список GUID свойств приведен в таблице 3.7.

**Таблица 3.7.** GUID свойств

<b>REFGUID</b>	<b>Описание</b>
<b>DIPROP_AUTOCENTER</b>	Указывает, является ли устройство самоцентрируемым
<b>DIPROP_AXISMODE</b>	Задаёт обсуждавшийся ранее режим осей
<b>DIPROP_BUFFERSIZE</b>	Определяет размер входного буфера
<b>DIPROP_CALIBRATIONMODE</b>	Указывает должен ли DirectInput получать от устройства калиброванные данные. По умолчанию все данные калиброваны
<b>DIPROP_DEADZONE</b>	Размер мертвой зоны, в которой не регистрируются изменения. Может изменяться от 0 (отсутствует) до 10000 (100 процентов)
<b>DIPROP_RANGE</b>	Устанавливает минимальное и максимальное значение диапазона
<b>DIPROP_SATURATION</b>	Значение, определяющее максимальный диапазон устройства. Может находиться в диапазоне от 0 до 10000. Например, значение 9500 означает, что устройство, такое как джойстик, отклоненное на 95 процентов заданного диапазона будет считаться полностью отклоненным

Структура **DIPROPHEADER**, которая используется в вызове **SetProperty**, определена следующим образом:

```
typedef struct {
    DWORD dwSize;           // Размер объемлющей структуры
    DWORD dwHeaderSize;     // Размер данной структуры
    DWORD dwObj;            // Параметр, который мы устанавливаем
    DWORD dwHow;            // Как мы устанавливаем значение
} DIPROPHEADER, *LPDIPROPHEADER
```

Поля **dwSize** и **dwHeaderSize** задают размеры структур в байтах, как описано в комментариях. Параметр **dwObj** может принимать множество различных значений, в зависимости от того, какое свойство мы устанавливаем. Например, чуть позже мы будем указывать значения **DIJOFS\_X** и **DIJOFS\_Y**, которые представляют ось X и ось Y джойстика, соответственно.

Для последнего параметра, **dwHow**, мы будем задавать значение **DIPH\_BYOFFSET**. DirectX SDK по этому параметру определяет, что будет указано смещение в текущем формате данных для того объекта, к которому мы хотим получить доступ. Это значит, что **DIOFS\_X** и **DIOFS\_Y** — это смещения устанавливаемых значений в текущем формате данных.

Позднее в этой главе я покажу вам, как устанавливать свойства для джойстика.

## Захват устройства

Перед использованием любое устройство должно быть *захвачено* (*acquired*). Захват устройства обеспечивает вашей программе доступ к нему, независимо от того, использует ли ваша программа совместный с другими приложениями доступ, или получает полный контроль над устройством. Будьте бдительны — другие программы могут бороться за те же права и похитить у вас контроль. Чтобы исправить это вы должны заново захватить устройство для использования.

Как узнать, когда нужно захватывать устройство? Во-первых, всегда при создании интерфейса — вы должны выполнить захват перед его использованием. Во-вторых, в любое другое время, когда другая программа перехватывает контроль и DirectInput сообщает об этом вашей программе.

Захват устройства выполняет вызов **IDirectInputDevice8::Acquire**:

```
HRESULT IDirectInputDevice8::Acquire();
```

### ПРИМЕЧАНИЕ

Обратившись к документации DX SDK, вы увидите, что функции **IDirectInputDevice8** возвращают стандартный код ошибки **DIERR\_INPUTLOST**, сообщающий, что устройство необходимо захватить (поскольку доступ к нему был утрачен).

Когда вы закончите работу, необходимо отпустить устройство. Это работа функции **IDirectInputDevice8::Unacquire**:

```
HRESULT IDirectInputDevice8::Unacquire();
```

---

**ВНИМАНИЕ!**

Убедитесь, что завершив работу с устройством вы вызываете для него функцию **Unacquire**. Отсутствие этого вызова может привести к зависанию системы.

---

## Опрос устройства

Опрос подготавливает устройство и, в ряде случаев, считывает для вас данные устройства, поскольку этот процесс может оказаться критичным по времени. Например, в случае джойстика, компьютер должен послать устройству импульс, чтобы чуть позже прочитать его данные. Это справедливо для некоторых джойстиков и не требуется для мышей и клавиатур.

Это не должно отвращать нас от использования опроса, поскольку с ним ядро кода будет более общим и сможет работать с любым устройством. Не беспокойтесь — опрос устройства, для которого он не требуется, не производит никакого эффекта. В конце концов, код будет более ясным.

Опрос устройства выполняется через **IDirectInputDevice8::Poll**:

```
HRESULT IDirectInputDevice8::Poll();
```

## Чтение данных

Наконец! Вы достигли цели — завершающего чтения данных устройства, которое выполняет функция **IDirectInputDevice8::GetDeviceState**. Вы должны передать этой функции буфер данных, в котором будет сохранена информация устройства для использования в вашей программе. Как вы вскоре увидите, у разных устройств эти данные отличаются.

Вот прототип функции:

```
HRESULT IDirectInputDevice8::GetDeviceState(  
    DWORD cbData,    // Размер буфера для хранения данных устройства  
    LPVOID lpvData); // Указатель на буфер для хранения  
                    // данных устройства
```

Независимо от устройства, для чтения данных применяется приведенный ниже фрагмент кода. Он учитывает, что вы можете потерять фокус устройства и его потребуется снова захватить. Вы должны передать указатель на буфер, размер которого достаточно большой для сохранения данных устройства, а также количество считываемых данных. В следующих разделах («DirectInput и клавиатура», «DirectInput и мышь» и «DirectInput и джойстик») я покажу вам, как использовать приведенную ниже функцию для получения данных каждого из упомянутых устройств.

```

BOOL ReadDevice(IDirectInputDevice8 *pDIDevice,
                void *DataBuffer, long BufferSize)
{
    HRESULT hr;

    while(1) {
        // Опрос устройства
        g_pDIDevice->Poll();

        // Чтение состояния
        if(SUCCEEDED(hr = g_pDIDevice->GetDeviceState(BufferSize,
                                                         (LPVOID)DataBuffer)))
            break;

        // Возвращаем нераспознанную ошибку
        if(hr != DIERR_INPUTLOST && hr != DIERR_NOTACQUIRED)
            return FALSE;

        // Вновь захватываем устройство и пытаемся
        // получить данные еще раз
        if(FAILED(g_pDIDevice->Acquire()))
            return FALSE;
    }
    // Сообщаем об успехе
    return TRUE;
}

```

## DirectInput и клавиатура

Здесь вы узнаете, как инициализировать и использовать клавиатуру. Приведенная ниже функция инициализации возвращает указатель на созданный объект **IDirectInputDevice8** или **NULL**, если произошел сбой. Вам надо только передать функции дескриптор родительского окна и ранее инициализированный объект **DirectInput**.

```

IDirectInputDevice8* InitKeyboard(HWND hWnd, IDirectInput8 *pDI)
{
    IDirectInputDevice8 *pDIDevice;

    // Создание объекта устройства
    if(FAILED(pDI->CreateDevice(GUID_SysKeyboard,
                               &pDIDevice, NULL)))
        return NULL;

    // Установка формата данных
    if(FAILED(pDIDevice->SetDataFormat(&c_dfDIKeyboard))) {
        pDIDevice->Release();
        return NULL;
    }

    // Установка режима кооперации
    if(FAILED(pDIDevice->SetCooperativeLevel(hWnd,
                                              DISCL_FOREGROUND | DISCL_NONEXCLUSIVE))) {
        pDIDevice->Release();
        return NULL;
    }
}

```

```
// Захватываем устройство для использования
if(FAILED(pDIDevice->Acquire())) {
    pDIDevice->Release();
    return NULL;
}

// Если все успешно, возвращаем указатель
return pDIDevice;
}
```

Функция **InitKeyboard** точно следует тому, о чем я говорил в этой главе. В следующем разделе, «DirectInput и мышь», вы увидите, что код инициализации мыши очень похож, и в результате я в ядре создам общую функцию для обоих устройств. Для чтения поступающих от клавиатуры данных вы можете использовать функцию **ReadData**.

Сперва вы должны понять, как клавиатура хранит данные. Вы должны предоставить массив из 256 байт, в котором каждый байт хранит состояние единственной клавиши. Это позволяет вам представлять 256 клавиш. Каждый байт хранит информацию о текущем состоянии клавиши — нажата она или нет. Чтобы определить состояние клавиши, проверьте старший бит (бит 7). Если он установлен — клавиша нажата; если сброшен — клавиша отпущена.

---

<b>ПРИМЕЧАНИЕ</b>	У каждой клавиши есть назначенный ей DirectInput макрос с префиксом <b>DIK_</b> . Так, клавиша <b>A</b> определена как <b>DIK_A</b> , клавиша <b>Esc</b> — как <b>DIK_ESCAPE</b> и т.д. Остальные макросы вы найдете в документации DX SDK или в файле <b>DInput.h</b> .
-------------------	--

---

Вот пример создания и чтения клавиатуры:

```
// Убедитесь в наличии инициализированных глобального
// объекта DirectInput и дескриптора родительского окна
// в переменных g_pDI и g_hWnd
IDirectInputDevice8 *pDIKeyboard;

// Буфер данных для хранения состояний клавиш
char KeyStateBuffer[256];

if((pDIKeyboard = InitKeyboard(g_hWnd, g_pDI)) != NULL) {
    // Читаем данные
    ReadData(pDIKeyboard, (void*)KeyStateBuffer, 256);
}
```

Вы можете создать макрос, упрощающий проверку того, нажата клавиша или отпущена. Этот макрос возвращает **TRUE**, если клавиша нажата, и **FALSE** — если отпущена.

```
#define KeyState(x) ((KeyStateBuffer[x] & 0x80) ? TRUE : FALSE)
```

Вот пример использования этого макроса:

```
if(KeyState(VK_LEFT) == TRUE) {
    // Нажата стрелка влево
}
```

## DirectInput и мышь

Следующая в очереди — мышь. Инициализация мыши практически полностью идентична инициализации клавиатуры, за исключением указания специфичных для мыши идентификатора и формата данных:

```
IDirectInputDevice8* InitMouse(HWND hWnd, IDirectInput8* pDI)
{
    IDirectInputDevice8 *pDIDevice;

    // Создаем объект устройства
    if(FAILED(pDI->CreateDevice(GUID_SysMouse, &pDIDevice, NULL)))
        return NULL;

    // Устанавливаем формат данных
    if(FAILED(pDIDevice->SetDataFormat(&c_dfDIMouse))) {
        pDIDevice->Release();
        return NULL;
    }

    // Задаем режим кооперации
    if(FAILED(pDIDevice->SetCooperativeLevel(hWnd,
        DISCL_FOREGROUND | DISCL_NONEXCLUSIVE))) {
        pDIDevice->Release();
        return NULL;
    }

    // Захватываем устройство для использования
    if(FAILED(lpDIDevice->Acquire())) {
        pDIDevice->Release();
        return NULL;
    }

    // Если все успешно, возвращаем указатель
    return lpDIDevice;
}
```

Вызов **IDirectInputDevice8::GetDeviceState** заполняет структуру данными о мыши, такими как относительное перемещение и состояние кнопок. Определение структуры **DIMOUSESTATE** выглядит так:

```
typedef struct {
    LONG lX;    // Относительное изменение координаты X
    LONG lY;    // Относительное изменение координаты Y
    LONG lZ;    // Относительное изменение координаты Z
    BYTE rgbButtons[4]; // Флаги нажатых клавиш
} DIMOUSESTATE, *LPDIMOUSESTATE;
```

Поскольку значения координат, сообщаемые в структуре **DIMOUSESTATE**, являются относительными, вам необходимо отслеживать абсолютные значения. Для этого вы создаете пару представляющих абсолютные координаты мыши глобальных переменных с именами **g\_MouseXPos** и **g\_MouseYPos**.

Взгляните на создание объекта и последующее чтение и обработку информации мыши:

```
// Убедитесь, что ранее инициализированы
// глобальный объект DirectInput g_pDI
// и дескриптор родительского окна g_hWnd
IDirectInputDevice8 *pDIDMouse;

// Координаты мыши
long g_MouseXPos = 0, g_MouseYPos = 0;

// Буфер данных для хранения состояния мыши
DIMOUSESTATE MouseState;

if((pDIDMouse = InitMouse(g_hWnd, g_pDI)) != NULL) {
    // Чтение данных
    ReadData(pDIDMouse, (void*)MouseState, sizeof(DIMOUSESTATE));

    // Обновление абсолютных координат
    g_MouseXPos += MouseState.lX;
    g_MouseYPos += MouseState.lY;
}
```

Для проверки состояния кнопок мыши можно использовать макрос, аналогичный тому, который применялся для проверки состояния клавиш клавиатуры:

```
#define MouseButtonState(x) ((MouseState.rgbButtons[x] & 0x80)
                             ? TRUE : FALSE)
```

Вот пример его использования:

```
if(MouseButtonState(0) == TRUE) {
    // нажата кнопка 0
}
```

## DirectInput и джойстик

Ну вот, и самое сложное для использования устройство. Самая трудная часть — это инициализация. Вы должны выполнить перечисление, чтобы обнаружить подключенные к системе джойстики. В процессе перечисления вы должны решить, какой джойстик будет использоваться и создать COM-объект для него. В этой книге мы рассмотрим только вариант с единственным джойстиком в системе.

```
// Убедитесь, что ранее инициализированы
// глобальный объект DirectInput g_pDI
// и дескриптор родительского окна g_hWnd
IDirectInputDevice8 *g_pDIDJoystick = NULL;

BOOL CALLBACK EnumJoysticks(LPCDIDEVICEINSTANCE pdInst,
                             LPVOID pvRef)
{
    HRESULT hr;
    g_pDIDJoystick = NULL;
```

Начинается перечисление очень просто. Вы создаете глобальный объект **IDirectInputDevice8** для использования джойстика. В начале перечисления вы присваиваете указателю на интерфейс значение **NULL**,

указывающее, что ничего не найдено. После перечисления вы проверяете, осталось ли значение указателя равным **NULL**, что свидетельствует о том, что джойстики не были инициализированы.

Что касается аргументов функции перечисления, то **pdInst** — это указатель на структуру **DIDEVICEINSTANCE**, содержащую сведения о перечисляемом в данный момент устройстве. Вы можете получить необходимый для создания интерфейса устройства GUID из поля **guidInstance** этой структуры.

Предоставляемый пользователем указатель **pvRef** в данном случае не нужен, поскольку дескриптор родительского окна является глобальной переменной. Вы, конечно, можете передавать в этом поле указатель на структуру, содержащую те же сведения, но я считаю, что здесь проще использовать глобальную переменную.

Следующий фрагмент кода следует общему шаблону создания интерфейса устройства, за исключением того, что он возвращает значение **DIENUM\_CONTINUE**, чтобы при возникновении ошибки перечисление продолжалось. В системе может быть несколько джойстиков, но здесь вы будете иметь дело только с первым обнаруженным:

```
// Создаем объект устройства, используя глобальный объект DirectInput
if (FAILED(g_pDI->CreateDevice(pdInst->guidInstance,
                              &g_pDIDJoystick, NULL)))
    return DIENUM_CONTINUE;

// Устанавливаем формат данных
if (FAILED(g_pDIDJoystick->SetDataFormat(&c_dfDIJoystick))) {
    g_pDIDJoystick->Release();
    g_pDIDJoystick = NULL;
    return DIENUM_CONTINUE;
}

// Устанавливаем уровень кооперации
if (FAILED(g_pDIDJoystick->SetCooperativeLevel(hWnd,
        DISCL_FOREGROUND | DISCL_NONEXCLUSIVE))) {
    g_pDIDJoystick->Release();
    g_pDIDJoystick = NULL;
    return DIENUM_CONTINUE;
}
```

Теперь необходимо установить свойства устройства. В это входит задание диапазона значений для осей джойстика, а также установка мертвой зоны. Требуемые значения заносятся в структуру **DIPROPRANGE**, которая применяется для установки диапазона в вызове **IDirectInputDevice8::SetProperty**.

Определение структуры **DIPROPRANGE** выглядит так:

```
typedef struct DIPROPRANGE {
    DIPROPHEADER diph; // Обсуждавшийся ранее DIPROPHEADER
    LONG         lMin;  // Минимальное значение диапазона (по оси X или Y)
    LONG         lMax;  // Максимальное значение диапазона (по оси X или Y)
} DIPROPRANGE, *LPDIPROPRANGE;
```



---

<b>ПРИМЕЧАНИЕ</b>	Структура <b>DIPROPRANGE</b> полезна, когда необходимо установить свойство, являющееся диапазоном допустимых значений (например, от 10 до 200 или от -1000 до 20).
-------------------	--

---

Чтобы использовать структуру **DIPROPRANGE** нужно сначала объявить ее экземпляр и инициализировать его:

```
DIPROPRANGE dipr;

// Сначала очищаем структуру
ZeroMemory(&dipr, sizeof(DIPROPRANGE));

dipr.diph.dwSize      = sizeof(dipr);
dipr.diph.dwHeaderSize = sizeof(dipr);
```

Теперь вы задаете значение поля **dipr.diph.dwObj**, которое указывает для какой оси (X или Y) вы задаете диапазон значений, для чего применяются макросы **DIJOFS\_X** и **DIJOFS\_Y** соответственно. Давайте начнем с оси X, а затем перейдем к оси Y.

```
dipr.diph.dwObj = DIJOFS_X;
dipr.diph.dwHow = DIPH_BYOFFSET; // Смещение в формате данных
```

Первое свойство, которое вы устанавливаете, — диапазон значений для оси X. Устанавливаем минимальное и максимальное значение для диапазона от -1024 (крайняя левая позиция) до +1024 (крайняя правая позиция).

```
dipr.lMin = -1024;
dipr.lMax =  1024;
```

---

<b>ПРИМЕЧАНИЕ</b>	Теперь вы видите, что назначение структуры <b>DIPROPRANGE</b> — задание диапазонов; здесь мы задаем диапазон значений от -1024 до 1024, это значит, что использоваться будут любые значения, находящиеся между этими двумя числами.
-------------------	---

---

Теперь для установки диапазона значений оси X вызываем функцию **IDirectInputDevice8::SetProperty**:

```
if(FAILED(g_pDIDJoystick->SetProperty(DIPROP_RANGE,
                                       &dipr.diph))) {
    g_pDIDJoystick->Release();
    g_pDIDJoystick = NULL;
    return DIENUM_CONTINUE;
}
```

Теперь установим диапазон значений для оси Y. Просто изменим значение **dwObj** (на **DIJOFS\_Y**) и снова вызовем функцию установки свойства.

```
dipr.diph.dwObj = DIJOFS_Y;
if(FAILED(g_pDIDJoystick->SetProperty(DIPROP_RANGE,
                                       &dipr.diph))) {
    g_pDIDJoystick->Release();
    g_pDIDJoystick = NULL;
    return DIENUM_CONTINUE;
}
```

Ну вот! Осталось только установить диапазон мертвой зоны, чтобы небольшие перемещения джойстика не вызывали дрожания. Установка мертвой зоны слегка отличается от установки диапазонов осей, поскольку здесь вам надо указать единственное значение — процент, который занимает мертвая зона в диапазоне значений оси. Вы можете спокойно устанавливать мертвую зону, занимающую около 15 процентов диапазона; это значит, что пользователь должен отклонить рукоятку не менее чем на 15 процентов в любом направлении, чтобы перемещение было зафиксировано.

Чтобы установить размер мертвой зоны мы используем структуру **DIPROPDWORD**, которая во многом похожа на структуру **DIPROPRANGE**, и содержит структуру **DIPROPHEADER** и значение типа **DWORD**:

```
typedef struct DIPROPDWORD {
    DIPROPHEADER diph;    // Обсуждавшийся ранее заголовок свойства
    DWORD        dwData;  // Значение DWORD для свойства
} DIPROPDWORD, *LPDIPROPDWORD;
```

Чтобы использовать структуру **DIPROPDWORD** для установки мертвой зоны, укажите в заголовке свойства, что работаете с осью X (воспользовавшись макросом **DIJOFS\_X**), задайте процентное значение (в диапазоне от 0 для 0% до 10000 для 100%) и установите свойство с помощью вызова **SetProperty**:

```
dipdw.diph.dwObj = DIJOFS_X; // Ось X
dipdw.dwData     = 1500;     // Приблизительно 15% диапазона
if(FAILED(g_pDIDJoystick->SetProperty(DIPROP_DEADZONE,
                                       &dipdw.diph))) {
    g_pDIDJoystick->Release();
    g_pDIDJoystick = NULL;
    return DIENUM_CONTINUE;
}
```

#### ПРИМЕЧАНИЕ

Чтобы вычислить значение, которое надо указать в **dipdw.dwData**, умножьте желательный процент на 100. Например, 15% = 15 × 100 = 1500.

Повторите то же самое для оси Y (используя макрос **DIJOFS\_Y**):

```
dipdw.diph.dwObj = DIJOFS_Y; // Ось Y
if(FAILED(g_pDIDJoystick->SetProperty(DIPROP_DEADZONE,
                                       &dipdw.diph))) {
    g_pDIDJoystick->Release();
    g_pDIDJoystick = NULL;
    return DIENUM_CONTINUE;
}
```

Теперь устройство инициализировано, и пришло время захватить устройство и завершить перечисление:

```
// Захватываем устройство для использования
if(FAILED(g_pDIDJoystick->Acquire())) {
    g_pDIDJoystick->Release();
    g_pDIDJoystick = NULL;
    return DIENUM_CONTINUE;
}
```

```
// Останавливаем перечисление
return DIENUM_STOP;
} // Конец функции
```

Теперь переменная **g\_pDIDJoystick** содержит либо указатель на созданный объект, либо равна **NULL**, если джойстик не инициализирован. Если объект устройства был инициализирован, то вы можете читать информацию точно так же, как это делалось для клавиатуры и мыши, но используя в функции **ReadData** структуру **DIJOYSTATE**. Вот как выглядит структура **DIJOYSTATE**:

```
typedef struct DIJOYSTATE {
    LONG lX; // Абсолютное значение координаты X
    LONG lY; // Абсолютное значение координаты Y
    LONG lZ; // Абсолютное значение координаты Z
    LONG lRx; // Вращение по X
    LONG lRy; // Вращение по Y
    LONG lRz; // Вращение по Z
    LONG rgdSlider[2]; // Значения движков
    DWORD rgdwPOV[4]; // Значения POV
    BYTE rgbButtons[32]; // Флаги кнопок (для 32 кнопок)
} DIJOYSTATE, *LPDIJOYSTATE;
```

Теперь, когда функция перечисления готова, вы можете инициализировать джойстик с помощью следующего кода:

```
g_pDI->EnumDevices(DIDEVTYPE_JOYSTICK, EnumJoysticks,
                  NULL, DIEDFL_ATTACHEDONLY);
if(g_pDIDJoystick == NULL) {
    // Джойстик не инициализирован
}
```

Для чтения данных джойстика вызовите функцию **ReadData** со структурой **DIJOYSTATE**:

```
DIJOYSTATE JoystickState;
ReadData(g_pDIDJoystick, (void*)JoystickState, sizeof(DIJOYSTATE));
```

Вы можете получить значения осей непосредственно из структуры **JoystickState**, поскольку координаты являются абсолютными. Здесь не надо отслеживать относительные перемещения:

```
JoystickX = JoystickState.lX;
JoystickY = JoystickState.lY;
```

Кроме того, можно применять уже рассматривавшийся ранее макрос для чтения состояния кнопок:

```
#define JoystickButtonState(x) ((JoystickState.rgbButtons[x] & 0x80)
                               ? TRUE : FALSE)
```

## Заканчиваем с вводом

Информация в этой главе, которая сосредотачивалась на трех самых распространенных устройствах ввода — клавиатуре, мыши и джойстике, — должна подготовить вас к знакомству с миром устройств ввода. Технологии движутся вперед и появляются новые устройства ввода, которые вы, вероятно, захотите использовать в своих игровых проектах.

Подумайте о возможностях — шлемы виртуальной реальности, костюмы с обратной связью и даже сканеры мозговых волн — и все совместимо с DirectInput! В таком мире вы легко сможете использовать любые устройства, следуя технике, которую изучили в этой главе.

### Программы на CD-ROM

Программы, демонстрирующие обсуждавшийся в этой главе код, находятся на прилагаемом к книге CD-ROM. В папке \BookCode\Chap03\ вы найдете следующие программы:

**Shell** — базовое приложение, которое инициализирует для вас DirectInput и устройства. Местоположение: \BookCode\Chap03\Shell\.

**Enum** — программа, использующая пример перечисления из этой главы для отображения всех подключенных к системе устройств ввода. Местоположение: \BookCode\Chap03\Enum\.

**Keyboard** — программа создает интерфейс клавиатуры и читает данные из него. Местоположение: \BookCode\Chap03\Keyboard\.

**Mouse** — программа создает интерфейс мыши и читает данные из него. Местоположение: \BookCode\Chap03\Mouse\.

**Joystick** — программа создает интерфейс джойстика и читает данные из него. Местоположение: \BookCode\Chap03\Joystick\.



# Глава 4

## Воспроизведение звуков и музыки с DirectX Audio и DirectShow

Музыка укрощает диких зверей, и найдется комбинация звуков и музыки, которая успокоит нас. Да и вообще, разве можно получить удовольствие от игры без звука? Можете ли вы представить себе современный боевик, такой как «Матрица», в виде немого кино! Никаких громких взрывов, звуков ударов пуль и разговоров актеров — нет всего того, что делает кино полностью мультимедийным окружением.

Ваша игра заслуживает того же уровня качества, что и фильмы, которые вы смотрите, включая музыку и звуковые эффекты. Вы сможете сделать все это с помощью DirectX Audio и DirectShow — дуэтом мультимедийных компонентов Microsoft DirectX.

В этой главе вы узнаете о следующих вещах:

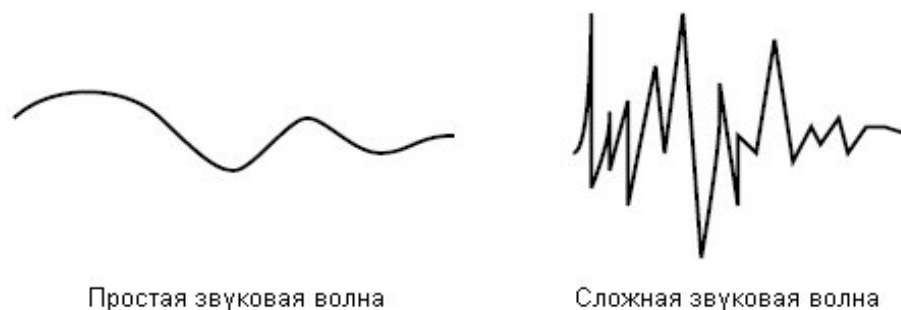
- Основные свойства звука.
- Цифровые форматы для звуков и музыки.
- Как использовать DirectX Audio в ваших проектах.
- Как воспроизводить звуки, используя DirectSound.
- Как заставить DirectMusic петь.
- Как воспроизводить MP3, используя DirectShow.

### Основы звука

Издает ли шум дерево, которое падает в лесу, где его никто не может услышать? Хотя часто этот вопрос считают провокационным, он имеет смысл. Звук — это просто волны идущие через среду, такую как воздух. С практической точки зрения, эти волны ничто, пока они не достигнут уха и не воздействуют на барабанную перепонку, воспроизводя звук.

Большинство природных звуков представляют собой «чистую» волну, для которой вы можете ясно видеть шаблон, которому она следует.

*Синусоидальная волна (sine wave)* — это хороший пример «чистой» волны; ее спады и подъемы чередуются по постоянному шаблону. Другие волны, такие как голос человека, более сложны — спады и подъемы чередуются очень быстро, не следуя какому-либо шаблону. На рис. 4.1 показаны две различных звуковых волны.



**Рис. 4.1.** Звуковые волны могут иметь простую, чистую форму или могут быть очень сложными

У каждого звука есть уникальные свойства, такие как амплитуда (уровень громкости) и частота. Вы можете в течение времени записывать свойства звуков, чтобы сохранить их в цифровой форме и потом воспроизвести.

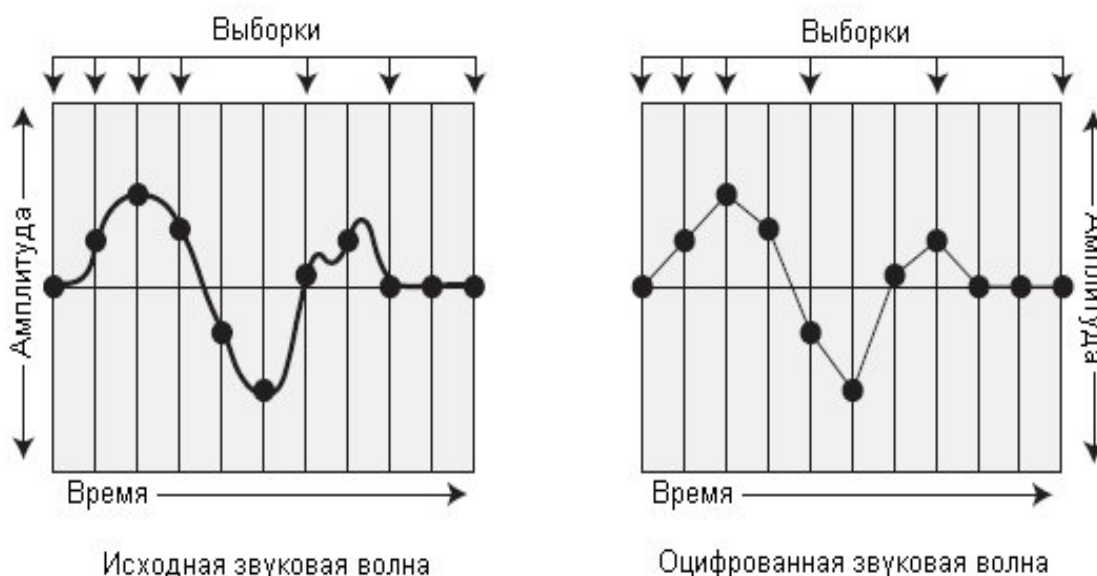
Кроме того, вы можете использовать эти звуки музыки, например, для того, чтобы играть композиции на вашем компьютере. Записав звуки, скажем, фортепьяно и скрипки, вы можете практически неотличимо воспроизводить реальные песни. Вообразите, что кто-то играет пьесу Моцарта, и нельзя отличить — реальный это оркестр или его компьютерный двойник!

## Цифровая звукозапись

Основы цифровой записи и воспроизведения звуков не слишком сложны для понимания. В общем случае вы берете звук, скажем, продолжительностью в две секунды, и исследуете звуковую волну от начала до конца с определенной периодичностью (*частотой выборки* или *частотой дискретизации, sampling rate*), которая измеряется в Герцах (Гц).

Скажем, вы хотите выполнить запись двухсекундного звука с частотой дискретизации 11 025 Гц, значит звук будет разделен на 22 050 фрагментов (по 11 025 фрагментов в секунде). Эти фрагменты называются *выборками (samples)*. Для каждой выборки вы определяете амплитуду звуковой волны в месте нахождения выборки, измеряя таким образом уровень громкости в определенный момент времени. Вы записываете полученное значение и переходите к следующей выборке. Записав уровень громкости для каждой выборки волны, вы получите цифровое представление звука.

На рис. 4.2 показана звуковая волна, разбитая на несколько выборок. При оцифровке звук теряет свою «волновую» форму. Оцифрованная волна не сохраняет форму оригинальной волны, поскольку для точного воссоздания оригинальной волны количества выборок недостаточно.



**Рис. 4.2.** Оригинальная волна на рисунке слева разбивается на несколько выборок. Вы используете эти выборки для создания цифрового представления звука, показанного на рисунке справа

Ясно, что чем выше частота дискретизации, использовавшаяся при захвате звуковой волны, тем точнее вы можете воссоздать исходный звук. «Какая частота дискретизации лучше» спрашиваете вы? У меня нет точного ответа для вас, это зависит от необходимого вам уровня качества, и обычно используются значения 8 000 Гц, 11 025 Гц, 22 050 Гц и 44 100 Гц. Последнее значение используется при записи музыки на компакт-диски и, предполагается, что оно обеспечивает наилучшее качество.

Следующая вещь, которую необходимо понимать — *хранение (storage)*. Захватывая выборку из волны вы получаете число, величина которого представляет амплитуду. Вы можете выбрать, хранить ли это значение как 8-разрядное, или как 16-разрядное. Это означает, что вы можете сохранять 256 уровней при использовании 8-разрядной выборки, или 65 536 уровней при использовании 16-разрядной выборки.

Выбирайте большее значение, чтобы у вас было больше уровней для репликации. 8-разрядные выборки должны использоваться только для низкокачественных звуков. Всегда старайтесь использовать 16-разрядный звук; даже несмотря на то, что его размер в два раза больше — он того стоит.

И, наконец, вы должны решить сколько каналов использовать. *Стереофонические* звуки (те, которые по-разному звучат с левой и с правой сторон) используют два канала. *Монофонические* звуки используют только один канал; следовательно со всех сторон слышен один и тот же звук.

Вы можете хранить цифровые звуки различными способами и использовать для них различные методы шифрования и сжатия. Я обычно храню их в несжатом формате волновых файлов Windows (.WAV), потому что это наиболее популярный (и лучше всего поддерживаемый) способ хранения звука.



Простейший волновой файл содержит единственный звук, которому предшествует описывающий данные заголовок. Win32 SDK предоставляет набор функций для работы с волновыми файлами, так что имея дело с ними вы можете обратиться к документации за дополнительной информацией.

## Музыкальное безумие

Музыка принимает множество форм, и вы можете записывать и воспроизводить ее различными способами. Технологические достижения позволили нам наслаждаться музыкой, которая использует цифровые звуки, что увеличивает наш слушательский опыт.

Музыка — ни что иное, как последовательность нот, воспроизводимых и останавливаемых в разное время с разными амплитудами. Для воспроизведения одной песни могут использоваться различные инструменты, но все они действуют одинаково — пользуясь нотами. Когда вы пишете песню на компьютере, то сохраняете ноты, а затем воспроизводите их, используя звуковую карту.

### ***MIDI***

Формат *MIDI* (идентифицируемый по расширению .MID) — это стандарт для хранения музыкальных партитур. Файлы MIDI содержат сообщения, или инструкции, необходимые для воспроизведения песни. Эти инструкции сообщают звуковому оборудованию что и когда делать, в том числе, когда и как начинать и завершать ноту, менять темп, использовать другой инструмент и т.д.

MIDI предоставляет до 128 стандартизованных инструментов, позволяя вам, например, составлять песню, используя инструмент 0 (который представляет пианино). Когда другие люди будут слушать песню, их устройство воспроизведения гарантирует, что играть будет указанный инструмент.

Песни делятся на треки, и каждый трек содержит ноты, воспроизводимые одним инструментом. В одном файле может быть до 128 треков, а длина их не ограничена. Это обещает самое настоящее музыкальное безумие!

MIDI-песни могут воспроизводить не только компьютеры; есть музыкальные инструменты с поддержкой MIDI, такие как синтезаторы. Верно. Вы можете подключить компьютер к такому музыкальному инструменту и получить устройство, воспроизводящее для вас инструментальные данные.

### ***DirectMusic***

Формат DirectMusic похож на формат MIDI; однако формат DirectMusic добавляет больше функциональных возможностей, таких как возможность создавать динамические музыкальные последовательности, генерируя согласованные уникальные музыкальные события. Во время

воспроизведения вы можете менять аккорды, темп и инструменты, создавая таким образом мощную систему.

Музыкальные *сегменты* (*segments*) DirectMusic (собственно файлы песен) используют расширение файлов .SGT. Связанные с сегментами файлы включают *оркестры* (*band*, расширение .BND), содержащие информацию об инструментах, *карты аккордов* (*chordmap*, расширение .CDM), содержащие аккорды для смены воспроизведения, *стили* (*style*, расширение .STY) для создания стилей воспроизведения, и *шаблоны* (*template*, расширение .TPL) для конструирования музыкальных сегментов.

Когда вы конструируете песню DirectMusic, все необходимые файлы обрабатываются DirectMusic автоматически (поэтому я ссылаюсь на песни DirectMusic, как на *родной* (*native*) формат). Это означает, что вы будете иметь дело только с файлами сегментов.

## MP3

MIDI достаточно мощный в своей области, и единственный недостаток — необходимость использовать синтез для репликации инструментов во время воспроизведения. Поскольку MIDI хранит только информацию о нотах, компьютер должен воспроизводить ноты, используя звуковое оборудование. Хотя достижения производящей аппаратуры индустрии сделали возможным очень точное воспроизведение звуков инструментов, существуют ситуации, когда вам нужна не синтезированная, а реальная записанная музыка. Здесь на сцену выходит MP3 (это сокращение для *MPEG Layer 3*).

Файл формата MP3, определяемый по расширению файла .MP3, — это звуковой файл, чем-то похожий на файл .WAV. Главное различие в том, что файлы .MP3 могут сжимать звуковые данные, уменьшая их размер без заметного ухудшения качества. Альбом с компакт-диска, занимающий обычно около 700 мегабайт необработанных звуковых данных, может быть сжат до размера в 40 мегабайт! Так что вместо прослушивания синтезированных музыкальных инструментов, вы можете выбрать реальную музыку за счет небольшого увеличения объема занимаемой памяти и возросшей нагрузки на процессор.

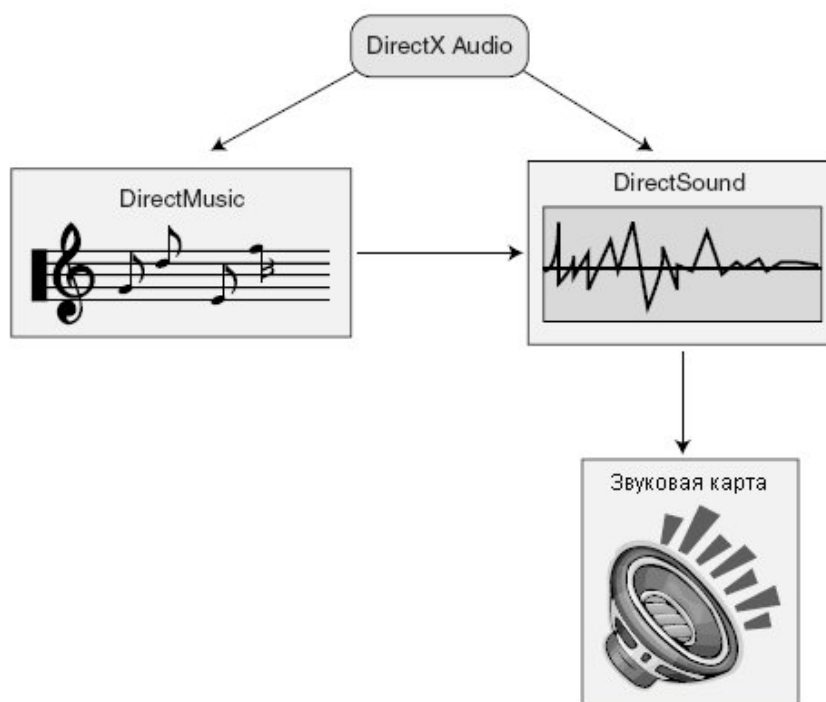
Возможность воспроизводить файлы MP3 предоставляет DirectShow — мультимедийный источник энергии Microsoft DirectX. Хотя DirectShow может воспроизводить практически любой мультимедийный файл, который вы ему предоставите, включая звуковые и видеозаписи, я буду использовать DirectShow только для воспроизведения звуковых файлов.

## Знакомство с DirectX Audio

DirectX Audio состоит из компонентов DirectX DirectSound и DirectMusic. Из этих компонентов DirectMusic наиболее изменился после версии DirectX 8. DirectSound в большинстве версий DirectX оставался практически неизменным. Фактически, большинство связанных со звуком компонентов,

которые я буду использовать, относятся к версии 8, так что не пугайтесь, несмотря на то, что мы в действительности используем DirectX 9!

DirectSound — основной компонент, используемый для воспроизведения цифрового звука. DirectMusic обрабатывает все звуковые форматы, — в том числе MIDI, родной формат DirectMusic и WAV-файлы, — и отправляет их DirectSound для цифрового воспроизведения (как показано на рис. 4.3). Это означает, что для MIDI во время воспроизведения вы можете использовать цифровые записи инструментов.



**Рис. 4.3.** *DirectX Audio использует отдельные компоненты DirectMusic и DirectSound, но позволяет DirectMusic использовать DirectSound для синтеза звуков инструментов и их воспроизведения на вашей звуковой карте*

Теперь, получив базовые знания, вы готовы узнать как использовать DirectSound и DirectMusic, о чем я буду говорить в оставшихся разделах этой главы.

## Использование DirectSound

Хотя на первый взгляд DirectSound может показаться сложным, в реальности он не столь труден для понимания и использования. Вы создаете COM-объект, являющийся интерфейсом для звукового оборудования. Этот COM-объект дает вам возможность создавать индивидуальные звуковые буферы (называемые *вторичными звуковыми буферами*, *secondary sound buffer*) для хранения звуковых данных.

Данные из этих звуковых буферов смешиваются вместе в главном буфере микширования (называемом *первичный звуковой буфер*, *primary sound buffer*) и воспроизводятся в указанном вами звуковом формате. Эти форматы воспроизведения могут отличаться по частоте, количеству каналов

и разрядности выборки. Допустимые частоты — 8 000 Гц, 11 025 Гц, 22 050 Гц и 44 100 Гц (CD-качество).

Для количества каналов есть два варианта: один канал для монофонического звука или два канала для стереофонического. Параметры разрядности также ограничиваются двумя вариантами: 8 бит для низкогокачественного воспроизведения звука и 16 бит для высококачественного воспроизведения. По умолчанию, если вы не выполняли настройку вручную, для первичного звукового буфера DirectSound установлены следующие параметры: 22 050 Гц, 8-разрядная выборка, стереофонический звук.

Вы можете модифицировать звуковые каналы для воспроизведения с другой частотой (меняя высоту звука), менять в ходе воспроизведения громкость и панорамирование звука, и даже заикливать звук. Но это не все — звуки могут воспроизводиться в виртуальном трехмерном окружении, имитирующем перемещающиеся вокруг вас реальные звуки.

Ваша задача — взять звуки и поместить их в звуковые буферы. Для очень больших звуков вы можете использовать метод потокового воспроизведения при котором загружается небольшой фрагмент звуковых данных, а когда его воспроизведение закончится, вы подгружаете в звуковой буфер следующий фрагмент данных. Этот процесс продолжается, пока весь звуковой файл не будет воспроизведен.

Вы реализуете потоковое воспроизведение, устанавливая позицию в звуковом буфере при достижении которой приложению будет отправлен сигнал о том, что пришло время обновить звуковые данные. Этот процесс передачи сигналов об обновлении называют *уведомлением (notification)*. Нет никаких ограничений на то, сколько буферов могут воспроизводиться одновременно, но вам следует следить, чтобы их было мало, поскольку каждый буфер создает дополнительную нагрузку на процессор и память.

В действительности работать с DirectSound не так уж трудно. Фактически, в этой книге мы будем работать только с тремя интерфейсами, перечисленными в таблице 4.1.

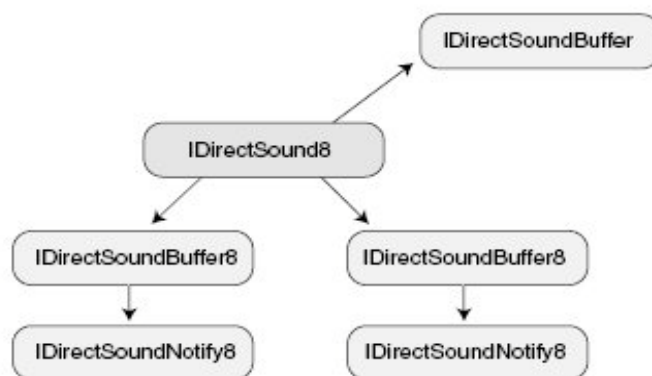
**Таблица 4.1.** COM-интерфейсы DirectSound

<b>Интерфейс</b>	<b>Описание</b>
<b>IDirectSound8</b>	Интерфейс главного объекта DirectSound.
<b>IDirectSoundBuffer8</b>	Объект первичного и вторичного звуковых буферов. Хранит данные и управляет воспроизведением.
<b>IDirectSoundNotify8</b>	Объект уведомления. Сообщает приложению о достижении заданной позиции в звуковом буфере.

**ПРИМЕЧАНИЕ**

Чтобы использовать в вашем проекте DirectSound и DirectMusic надо включить заголовочные файлы dsound.h и dmusic.h и добавить в список компоновки файл библиотеки dsound.lib. Также хорошо бы указать при компоновке и библиотеку dxguid.lib, поскольку в ней определены некоторые полезные элементы, которые используются DirectSound.

На рис. 4.4 показаны взаимоотношения между этими объектами. **IDirectSound8** — это главный интерфейс из которого вы создаете звуковые буфера (**IDirectSoundBuffer8**). Затем звуковой буфер может создать собственный интерфейс уведомления (**IDirectSoundNotify8**), который вы используете для отметки позиции в звуковом буфере по достижении которой будет отправлено уведомление. Интерфейс уведомлений полезен при потоковом воспроизведении звука.



*Рис. 4.4. Вы получаете звуковой буфер через объект IDirectSound8. Объект IDirectSoundNotify8 создается через его родителя — объект IDirectSoundBuffer8*

## Инициализация DirectSound

Перед тем, как делать что-либо еще, вам необходимо включить заголовочный файл dsound.h и указать в списке компоновки библиотеку dsound.lib. После этого можно приступать к первому этапу использования DirectSound — созданию объекта **IDirectSound8**, который является главным интерфейсом, представляющим звуковое оборудование. Делается это с помощью функции **DirectSoundCreate8**:

```

HRESULT WINAPI DirectSoundCreate8(
    LPCGUID lpcGuidDevice, // Укажите NULL
                           // (звуковое устройство по умолчанию)
    LPDIRECTSOUND8 *ppDS8, // Создаваемый объект
    LPUNKNOWN pUnkOuter); // NULL - не используется
  
```

**ПРИМЕЧАНИЕ**

Функция **DirectSoundCreate8**, подобно всем другим функциям DirectSound, возвращает **DS\_OK**, если завершилась успешно, или код ошибки в противном случае. Чтобы контроль ошибок был проще можно для проверки возвращаемого значения использовать макросы **FAILED** и **SUCCEEDED**.

Используя функцию **DirectSoundCreate8** и глобальный экземпляр объекта **IDirectSound8**, можно инициализировать объект звуковой системы следующим образом:

```
IDirectSound8 *g_pDS; // Глобальный объект IDirectSound8

if(FAILED(DirectSoundCreate8(NULL, &g_pDS, NULL))) {
    // Произошла ошибка
}
```

## Установка уровня кооперации

Следующий этап инициализации — установка *уровня кооперации* (*cooperative level*) объекта **IDirectSound8**. Вы используете уровень кооперации, чтобы определить как будет осуществляться совместное с другими приложениями использование ресурсов звуковой карты. Хотите ли вы, чтобы карта была полностью в вашем распоряжении, не позволяя никому другому выполнять воспроизведение; или наоборот — хотите разрешить совместный доступ? Или вам нужен специальный формат воспроизведения, который не совпадает с установленным по умолчанию?

**Таблица 4.2.** Уровни кооперации DirectSound

Уровень	Макрос	Описание
Нормальный	<b>DSSCL_NORMAL</b>	Обычный уровень; позволяет всем программам обращаться к звуковой карте, используя формат воспроизведения по умолчанию: 8 бит, 11025 Гц, 1 канал (моно). Этот формат не может быть изменен.
Приоритетный	<b>DSSCL_PRIORITY</b>	То же, что и нормальный, но позволяет вам менять формат воспроизведения.
Монопольный	<b>DSSCL_EXCLUSIVE</b>	Монопольное использование звуковой карты; никакие другие приложения не могут использовать звуковое устройство, пока ваша программа активна. Вы задаете формат воспроизведения.
Первичная запись	<b>DSSCL_WRITEPRIMARY</b>	Профессиональный уровень, предоставляющий вам полный контроль над системой. Вы получаете доступ только к первичному звуковому буферу (вторичные звуковые буферы отключены). Режим предназначен для программистов, которые хотят в своем коде реализовать собственный алгоритм Микширования, поэтому в дальнейшем я не буду его рассматривать.

Установка уровня кооперации — это работа функции **IDirectSound8::SetCooperativeLevel**. Есть четыре возможных уровня кооперации, и они перечислены в таблице 4.2. Каждому соответствует собственный макрос, определенный в DirectSound.

Вот прототип функции **IDirectSound8::SetCooperativeLevel**:

```
IDirectSound8::SetCooperativeLevel(  
    HWND hwnd, // Дескриптор родительского окна  
    DWORD dwLevel); // Уровень кооперации из таблицы 4.2
```

Какой уровень кооперации использовать? Вообще-то это зависит от типа создаваемого приложения. Для полноэкранных приложений используйте монопольный уровень. В других случаях я рекомендую приоритетный уровень. Будьте бдительны, когда используете уровень отличный от нормального — помните, что в таком случае вам надо задать формат воспроизведения. Я покажу вам, как это делается в следующем разделе, «Установка формата воспроизведения».

---

**СОВЕТ**

Я настоятельно рекомендую использовать приоритетный уровень кооперации предоставляющий вам контроль над первичным буфером, даже если вы не хотите менять формат воспроизведения. Благодаря этому вы сможете легко регулировать громкость, а также управлять панорамированием во время воспроизведения.

---

А теперь пример установки приоритетного уровня кооперации с использованием ранее инициализированного объекта **IDirectSound8**:

```
// g_pDS = ранее инициализированный объект IDirectSound8  
// hWnd = ранее инициализированный дескриптор родительского окна  
  
if(FAILED(g_pDS->SetCooperativeLevel(hWnd, DSSCL_PRIORITY))) {  
    // Произошла ошибка  
}
```

## Установка формата воспроизведения

Последний этап инициализации DirectSound, выполняемый только если вы используете уровень кооперации, отличный от нормального, — захват управления первичным звуковым буфером и установка формата воспроизведения для системы. Процесс делится на две части: сначала вы используете объект **IDirectSound8** для создания интерфейса буфера, а затем используете интерфейс для модификации формата.

### **Создание объекта первичного звукового буфера**

Объект **IDirectSoundBuffer** представляет первичный звуковой буфер. Здесь не требуется интерфейс восьмой версии, поскольку система микширования в разных версиях DX не менялась. Звуковой буфер (как

первичный, так и вторичный) создает функция **IDirectSound8::CreateSoundBuffer**, которая выглядит так:

```
HRESULT IDirectSound8::CreateSoundBuffer(
    LPCDSBUFFERDESC pcDSBufferDesc,    // Описание буфера
    LPDIRECTSOUNDBUFFER *ppDSBuffer,    // Создаваемый объект буфера
    LPUNKNOWN pUnkOuter);              // NULL - не используется
```

В параметре **pcDSBufferDesc** передается указатель на структуру **DSBUFFERDESC**, хранящую различную информацию о создаваемом буфере. Для первичного буфера вы не будете использовать все возможности, предоставляемые объектом звукового буфера, но тем не менее, взглянем на всю структуру целиком:

```
typedef struct {
    DWORD dwSize;                // Размер структуры
    DWORD dwFlags;                // Флаги, описывающие возможности буфера
    DWORD dwBufferBytes;          // Размер звукового буфера
    DWORD dwReserved;             // Не используется, установите 0
    LPWAVEFORMATEX lpwfxFormat;   // Формат воспроизведения
    GUID guid3DAlgorithm;         // GUID_NULL
                                // (Алгоритм 3D-воспроизведения)
} DSBUFFERDESC, *LPDSBUFFERDESC;
```

Назначение полей ясно из их названий, за исключением **lpwfxFormat**. Это указатель на структуру, описывающую формат воспроизведения создаваемого буфера. Поскольку пока мы не имеем дела с ней, оставим подробное описание на потом. Что касается **dwBufferBytes**, первичный звуковой буфер уже существует, так что присвойте **dwBufferBytes** значение 0.

---

<b>ПРИМЕЧАНИЕ</b>	DirectSound автоматически создает буфер данных, используемый как первичный звуковой буфер, поэтому нет никаких указаний расположен ли этот буфер в системной памяти или в памяти звуковой карты. Кроме того, установка формата воспроизведения для первичного звукового буфера выполняется немного по другому, поэтому вам надо присвоить указателю <b>lpwfxFormat</b> значение <b>NULL</b> .
-------------------	---

---

Сейчас вам придется иметь дело только с переменной **dwFlags**, которая содержит набор флагов, определяющих возможности создаваемого буфера. В таблице 4.3 перечислены все доступные для использования флаги.

Сейчас мы будем использовать только флаги **DSBCAPS\_PRIMARYBUFFER** и **DSBCAPS\_CTRLVOLUME**. Эти флаги сообщают DirectSound, что вы хотите создать интерфейс первичного звукового буфера и обеспечить возможность регулировать громкость воспроизведения. Позднее мы поговорим об оставшихся флагах.

---

<b>ВНИМАНИЕ!</b>	Некоторые флаги, такие как флаг контроля частоты, нельзя использовать для первичного буфера. Указание таких флагов приведет к возникновению ошибки при создании первичного буфера.
------------------	--

---



Таблица 4.3. Флаги для создания звукового буфера

<b>Флаг</b>	<b>Описание</b>
<b>DSBCAPS_CTRL3D</b>	У буфера есть трехмерные возможности.
<b>DSBCAPS_CTRLFREQUENCY</b>	Позволяет менять частоту в ходе воспроизведения буфера.
<b>DSBCAPS_CTRLFX</b>	Буфер позволяет обработку эффектов.
<b>DSBCAPS_CTRLPAN</b>	У буфера есть возможность позиционирования.
<b>DSBCAPS_CTRLPOSITIONNOTIFY</b>	Буфер может использовать уведомления.
<b>DSBCAPS_CTRLVOLUME</b>	Позволяет на лету регулировать громкость для буфера.
<b>DSBCAPS_GETCURRENTPOSITION2</b>	Позволяет запрашивать буфер о местонахождении позиции воспроизведения.
<b>DSBCAPS_GLOBALFOCUS</b>	Создает глобальный звуковой буфер; это значит, что воспроизводимый звук будет слышно даже если активна другая программа.
<b>DSBCAPS_LOCDEFER</b>	Позволяет буферу использовать аппаратные и программные ресурсы.
<b>DSBCAPS_LOCHARDWARE</b>	Заставляет использовать аппаратные ресурсы, такие как микшер и память звуковой карты.
<b>DSBCAPS_LOCSOFTWARE</b>	Заставляет использовать программные ресурсы, такие как программное микширование и системная память.
<b>DSBCAPS_MUTE3DATMAXDISTANCE</b>	Выключает трехмерный звук, если его источник находится слишком далеко от слушателя.
<b>DSBCAPS_PRIMARYBUFFER</b>	Делает создаваемый буфер первичным звуковым буфером. Используйте этот флаг только один раз и только тогда, когда устанавливаете уровень кооперации отличный от нормального.
<b>DSBCAPS_STATIC</b>	Если возможно, буфер будет размещен в памяти звуковой карты. Используйте флаг только для небольших звуков.
<b>DSBCAPS_STICKYFOCUS</b>	Заставляет буфер продолжать воспроизведение, когда пользователь переключается на другое приложение, не использующее DirectSound. Звук будет отключен, независимо от наличия флага.

Давайте пойдем дальше и создадим интерфейс первичного звукового буфера:

```

IDirectSoundBuffer g_pDSPrimary; // Глобальный доступ
DSBUFFERDESC      dsbd;          // Описание буфера

ZeroMemory(&dsbd, sizeof(DSBUFFERDESC)); // Обнуляем структуру

dsbd.dwSize        = sizeof(DSBUFFERDESC); // Устанавливаем размер
// структуры
dsbd.dwFlags        = DSBCAPS_PRIMARYBUFFER | DSBCAPS_CTRLVOLUME;
dsbd.dwBufferBytes = 0;                  // Не задаем размер буфера
dsbd.lpwfxFormat    = NULL;              // Не задаем формат
if(FAILED(g_pDS->CreateSoundBuffer(&dsbd,
    &g_pDSPrimary, NULL))) {
    // Произошла ошибка
}

```

## Установка формата

Теперь, когда вы получили возможность управлять первичным звуковым буфером, пришло время установить формат воспроизведения. У вас есть несколько вариантов, но я рекомендую использовать осмысленные значения, такие как 11 025 Гц, 16-разрядная выборка, моно или 22 050 Гц, 16-разрядная выборка, моно.

Выбирая формат, постарайтесь не использовать стереофонию. Она создает ненужную нагрузку на процессор, поскольку настоящие стереофонические звуки достаточно сложно записать. Также старайтесь всегда использовать 16-разрядную выборку, поскольку ее качество значительно выше, чем у 8-разрядной. Никогда не соглашайтесь на меньшее! Что касается частоты, то чем она выше — тем лучше, но не стоит выбирать частоту выше 22 050 Гц. Даже звуки с CD-качеством прекрасно воспроизводятся на 22 050 Гц без заметных потерь.

Все сказано, пойдем дальше. Вы устанавливаете формат воспроизведения через вызов **IDirectSoundBuffer::SetFormat**:

```

HRESULT IDirectSoundBuffer::SetFormat(
    LPCWAVEFORMATEX pcfxFormat);

```

### ВНИМАНИЕ!

Функция **SetFormat** должна вызываться только для объекта первичного звукового буфера.

Первый и единственный аргумент — это указатель на структуру **WAVEFORMATEX**, содержащую сведения о формате, который вы хотите установить:

```

typedef struct {
    WORD wFormatTag;          // Укажите WAVE_FORMAT_PCM
    WORD nChannels;           // 1 для моно, 2 для стерео
    DWORD nSamplesPerSec;     // Частота дискретизации
    DWORD nAvgBytesPerSec;    // Количество байт в секунду для формата
    WORD nBlockAlign;         // Размер данных выборки
    WORD wBitsPerSample;      // 8 или 16
    WORD cbSize;              // Не используется
} WAVEFORMATEX;

```

Вы должны без труда заполнить эту структуру, за исключением полей **nBlockAlign** и **nAvgBytesPerSec**. Переменная **nBlockAlign** — это количество байт, используемых для каждой выборки звука. Она инициализируется так:

```
nBlockAlign = (nBitsPerSample / 8) * nChannels;
```

Переменная **nAvgBytesPerSec** — это количество байт, необходимых для хранения одной секунды звука. Здесь надо принять во внимание частоту дискретизации и размер данных выборки, что приводит к следующей формуле:

```
nAvgBytesPerSec = nSamplesPerSec * nBlockAlign;
```

Теперь, вооружившись этими сведениями, пришло время попытаться применить их! Вот пример установки формата 22 050 Гц, 16 бит, моно:

```
// g_pDSPrimary = ранее инициализированный глобальный объект
//              первичного звукового буфера
WAVEFORMATEX wfex;
ZeroMemory(&wfex, sizeof(WAVEFORMATEX));

wfex.wFormatTag      = WAVE_FORMAT_PCM;
wfex.nChannels       = 1;           // моно
wfex.nSamplesPerSec  = 22050;       // 22050 Гц
wfex.wBitsPerSample  = 16;          // 16 бит
wfex.nBlockAlign     = (wfex.wBitsPerSample / 8) * wfex.nChannels;
wfex.nAvgBytesPerSec = wfex.nSamplesPerSec * wfex.nBlockAlign;

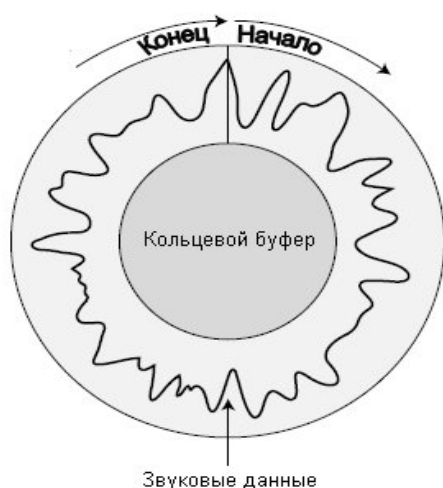
if(FAILED(g_pDSPrimary->SetFormat(&wfex))) {
    // Произошла ошибка
}
```

## Запуск первичного звукового буфера

Наконец вы получили контроль над звуковой системой, и готовы играть рок! Осталось только включить воспроизведение для первичного звукового буфера. Несмотря на то, что у вас еще нет звуков, включить воспроизведение буфера лучше сейчас, а потом не отключать его, пока вы не закончите использование звуковой системы. Запуск воспроизведения первичного буфера в начале приложения позволит сэкономить время процессора, которое иначе тратилось бы на запуски и остановки.

Перед тем, как показать запуск воспроизведения буфера, я должен рассказать вам о первичном звуковом буфере. Поскольку ресурсы памяти ограничены, особенно у звуковой карты, используемый буфер данных может быть любого размера (даже всего несколько сотен байт). По этой причине в качестве первичного и вторичных звуковых буферов используются *кольцевые буферы (circular buffer)*.

Пример кольцевого буфера показан на рис. 4.5. Несмотря на то, что буфер данных — это одномерный массив, его конец замыкается на начало. Это мощная техника, позволяющая экономить значительные объемы памяти.



*Рис. 4.5. Кольцевые буферы всегда замкнуты, соединяя начало с концом буфера, чтобы звуки могли постоянно ходить по кругу для непрерывного воспроизведения*

Воспроизводимые звуки смешиваются в первичном звуковом буфере, который является кольцевым буфером данных. Когда будет достигнут конец звукового буфера, звук за циклируется на начало буфера и его воспроизведение продолжается без паузы. Чтобы использовать возможность за цикливания буфера вы должны явно включить за цикленное воспроизведение; иначе воспроизведение буфера будет остановлено по достижении его конца.

Чтобы начать воспроизведение звукового буфера (с необязательной возможностью за цикленного воспроизведения), вы должны вызвать функцию звукового буфера **Play**, которая выглядит так:

```
HRESULT IDirectSoundBuffer8::Play(
    DWORD Reserved1, // Должен быть 0
    DWORD Priority,  // Приоритет микширования - используйте 0
    DWORD dwFlags); // Флаги воспроизведения
```

Единственный представляющий интерес аргумент — **dwFlags**, который может принимать два значения: 0 для однократного воспроизведения звукового буфера и остановки при достижении конца, и **DSBPLAY\_LOOPING** сообщаящий о необходимости постоянно при достижении конца возвращаться к началу буфера для непрерывного воспроизведения.

Для первичного звукового буфера вам практически всегда будет требоваться за цикленное воспроизведение, и вот как это можно сделать:

```
if(FAILED(g_pDSPrimary->Play(0, 0, DSBPLAY_LOOPING))) {
    // Произошла ошибка
}
```

Когда вы закончите работать с первичным звуковым буфером (и со звуковой системой в целом), надо остановить ее, вызвав функцию **IDirectSoundBuffer::Stop**, у которой нет аргументов:

```
if(FAILED(g_pDSPrimary->Stop())) {
    // Произошла ошибка
}
```

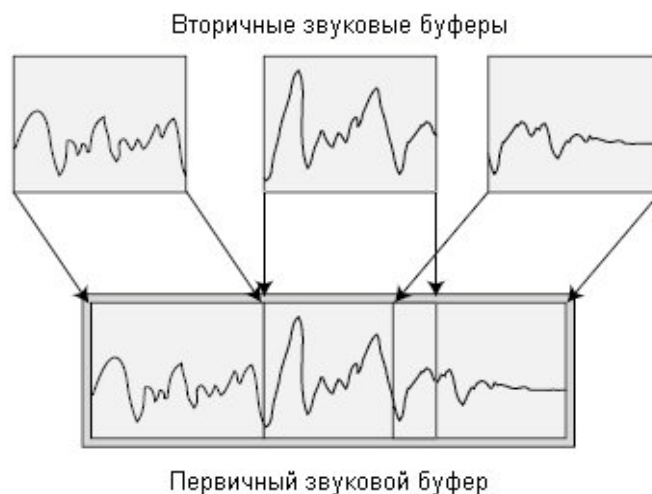
**ВНИМАНИЕ!**

Не надо останавливать первичный звуковой буфер, пока не будут остановлены все вторичные звуковые буферы. Когда воспроизводится вторичный звуковой буфер, автоматически запускается воспроизведение первичного буфера.

## Использование вторичных звуковых буферов

Далее на очереди создание вторичных звуковых буферов, содержащих реальные звуковые данные, которые вы хотите воспроизвести. Нет никаких ограничений на количество создаваемых вторичных звуковых буферов, и возможности DirectSound позволяют, если хотите, воспроизводить их все одновременно.

Вы достигаете этого, заполняя первичный звуковой буфер звуковыми данными, находящимися во вторичных звуковых буферах (этот процесс показан на рис. 4.6). Эти данные микшируются между собой, так что если записать звук в то место первичного звукового буфера, куда уже был записан другой звук, то будут воспроизводиться оба звука одновременно.



*Рис. 4.6. Перед началом воспроизведения вторичные звуковые буферы микшируются вместе в первичном звуковом буфере*

Вторичные звуковые буферы используют объект **IDirectSoundBuffer8**, очень похожий на объект **IDirectSoundBuffer**. Фактически, для создания восьмой версии интерфейса вы должны сперва создать объект **IDirectSoundBuffer** и запросить через него новый интерфейс.

В создании вторичных звуковых буферов есть одно отличие — вы должны при инициализации установить формат воспроизведения. Это значит, что буфер может использовать только один формат. Если вам надо сменить формат, освободите буфер и создайте другой.

Снова вы используете структуру **WAVEFORMATEX** для хранения формата и структуру **DSBUFFERDESC** для описания возможностей буфера.

Однако на этот раз вы указываете указатель на структуру **WAVEFORMATEX** внутри структуры **DSBUFFERDESC**.

Вот пример создания вторичного звукового буфера, использующего формат 22 050 Гц, 16 бит, моно. Я создаю буфер достаточный для хранения двухсекундного звука (поскольку сейчас я не буду помещать в него никаких реальных звуков), с возможностью регулировки громкости, позиционирования и частоты.

```
// g_pDS = ранее инициализированный объект IDirectSound8
IDirectSoundBuffer8 *g_pDSBuffer; // Глобальный объект 8 версии
IDirectSoundBuffer *pDSB;         // Локальный звуковой буфер

// Инициализируем структуру WAVEFORMATEX
WAVEFORMATEX wfex;
ZeroMemory(&wfex, sizeof(WAVEFORMATEX));

wfex.wFormatTag      = WAVE_FORMAT_PCM;
wfex.nChannels       = 1;           // моно
wfex.nSamplesPerSec  = 22050;      // 22050 Гц
wfex.wBitsPerSample  = 16;         // 16 бит
wfex.nBlockAlign     = (wfex.wBitsPerSample / 8) * wfex.nChannels;
wfex.nAvgBytesPerSec = wfex.nSamplesPerSec * wfex.nBlockAlign;

// Инициализируем структуру DSBUFFERDESC
DSBUFFERDESC dsbd;
ZeroMemory(&dsbd, sizeof(DSBUFFERDESC)); // Обнуляем структуру

dsbd.dwSize          = sizeof(DSBUFFERDESC); // Задаем размер
dsbd.dwFlags          = DSBCAPS_CTRLFREQUENCY | DSBCAPS_CTRLVOLUME |
                        DSBCAPS_CTRLPAN;
dsbd.dwBufferBytes    = wfex.nAvgBytesPerSec * 2; // 2 секунды
dsbd.lpwfxFormat      = &wfex;

// Создаем первую версию объекта
if(FAILED(g_pDS->CreateSoundBuffer(&dsbd, &pDSB, NULL))) {
    // Произошла ошибка
} else {
    // Получаем 8 версию интерфейса
    if(FAILED(pDSB->QueryInterface(IID_IDirectSoundBuffer8,
                                   (void**)&g_pDSBuffer))) {
        // Произошла ошибка - прежде чем делать
        // что-либо еще освобождаем первый интерфейс
        pDSB->Release();
    } else {
        // Освобождаем исходный интерфейс - все успешно!
        pDSB->Release();
    }
}
```

## Блокируй и загружай — загрузка звуковых данных в буфер

Великолепно! Звуковой буфер создан, и теперь мы готовы воспроизводить звуки! Осталась одна задача — поместить звуковые данные в буфер. В распоряжении звуковых буферов есть пара функций: **IDirectSoundBuffer8::Lock**, которая блокирует буфер звуковых

данных и возвращает указатель на область данных, и **IDirectSoundBuffer8::Unlock**, освобождающая ресурсы, используемые во время операции блокировки.

Блокируя буфер вы подготавливаете его для записи. Вы сообщаете буферу смещение (в байтах), начиная с которого вы хотите начать доступ, и количество байтов к которым необходим доступ. В свою очередь вы получаете два указателя на данные и две переменных, сообщающих сколько данных доступно.

Почему вы получаете два указателя и два размера? Потому что звуковой буфер является кольцевым и для блокировки требуемого количества байтов вам может потребоваться вернуться к началу буфера. Первый указатель — это запрошенная вами позиция, а первый размер — это количество байтов от этой позиции до конца буфера. Второй указатель обычно устанавливается на начало буфера, а второй размер — это оставшееся количество блокируемых байтов, которые простираются за конец звукового буфера. На рис. 4.7 показан пример буфера данных с двумя указателями и размерами.



**Рис. 4.7.** Буфер данных, размером 65536 байт, в котором блокируются для доступа 62000 байт. Первый указатель используется для доступа к 60000 байт, а второй указатель — для доступа к оставшимся 2000 байт

Вот как выглядит прототип функции блокировки:

```
HRESULT IDirectSoundBuffer8::Lock(
    DWORD    dwOffset,           // Смещение в буфере до начала
                                // блокируемой области
    DWORD    dwBytes,           // Количество блокируемых байтов
    LPVOID   *ppvAudioPtr1,     // Указатель на указатель на
                                // первый блок данных
    LPDWORD  *ppwAudioBytes1,   // Указатель на размер первого блока
    LPVOID   *ppvAudioPtr2,     // Указатель на указатель на
                                // второй блок данных
    LPDWORD  *ppwAudioBytes2,   // Указатель на размер второго блока
    DWORD    dwFlags);          // Флаги блокировки
```

Здесь следует обратить внимание на несколько вещей. Во-первых вам надо передать пару указателей (приведенных к типу **LPVOID\***), которые будут указывать на соответствующие данные в заблокированном звуковом

буфере. Помните, если размер блокируемого пространства превышает размер оставшегося фрагмента буфера и возникает эффект закольцовывания, то используется два указателя. Также вам надо предоставить два указателя на пару переменных типа **DWORD**, в которые будет занесено (функцией **Lock**) количество байт звукового буфера, к которым предоставляется доступ через два различных указателя на данные.

Во-вторых, посмотрите на переменную **dwFlags**, которая может принимать три значения: 0 для блокировки запрашиваемого фрагмента, **DSBLOCK\_FROMWRITECURCOR** для блокировки от текущей позиции записи и **DSBLOCK\_ENTIREBUFFER** для блокировки всего буфера, игнорируя указанные смещение и размер.

#### ВНИМАНИЕ!

Когда вы блокируете звуковой буфер, убедитесь, что разблокируете его как можно быстрее. Длительная блокировка буфера приводит к нежелательным эффектам. Также не пытайтесь заблокировать фрагмент данных, который воспроизводится в текущий момент.

#### СОВЕТ

Лучше всего присваивать переменной **dwFlags** значение 0 — это гарантирует, что будет заблокировано только указанное количество байтов, находящихся в заданном месте. Также, если вы не хотите использовать второй указатель или размер, присвойте соответствующим переменным значения **NULL** и 0.

Давайте, продолжим, заблокируем весь буфер данных и заполним его случайными числами:

```
// g_pDSBuffer = ранее инициализированный
//              вторичный звуковой буфер
char *Ptr;
DWORD Size;
if(SUCCEEDED(g_pDSBuffer->Lock(0, 0, (void**)&Ptr,
                               (DWORD*)&Size, NULL, 0, DSBLOCK_ENTIREBUFFER))) {
for(long i = 0; i < Size; i++)
    Ptr[i] = rand() % 65536;
```

К данному моменту вы закончили работу с буфером и готовы разблокировать его, чтобы освободить используемые в процессе ресурсы. Это делается с помощью функции **IDirectSoundBuffer8::Unlock**:

```
HRESULT IDirectSoundBuffer8::Unlock(
    LPVOID pvAudioPtr1,    // Первый указатель на данные
    DWORD  dwAudioBytes1,  // Первый размер
    LPVOID pvAudioPtr2,    // Второй указатель на данные
    DWORD  dwAutioBytes2); // Второй размер
```

Вам необходимо передать значения (а не указатели на них), полученные при блокировке буфера:

```
if(FAILED(g_pDSBuffer->Unlock((void*)Ptr, Size, NULL, 0))) {
    // Произошла ошибка
}
```



## Воспроизведение звукового буфера

Буфер разблокирован и данные загружены — настало время воспроизводить звук. Для воспроизведения вторичного звукового буфера вызывается та же функция, которая применялась для воспроизведения первичного буфера и была описана ранее в этой главе. Однако на этот раз вам не нужно заикливать звук, так что исключите флаг **DSBPLAY\_LOOPING**.

Мы подошли к одному примечательному различию — вы должны сообщить звуковому буферу с какой позиции внутри него надо начинать воспроизведение. Обычно первое воспроизведение звука вы начинаете с его начала. Однако остановка звука не сбрасывает позицию воспроизведения, поскольку вы можете включить паузу, а затем вызвать функцию воспроизведения снова, чтобы продолжить прослушивание с того места, на котором в последний раз остановились. Установка позиции воспроизведения легко выполняется с помощью следующей функции:

```
HRESULT IDirectSoundBuffer8::SetCurrentPosition(  
    DWORD dwNewPosition);
```

У функции всего один аргумент — смещение, с которого вы хотите начать воспроизведение звука. Оно должно быть выровнено на размер блока выборки, который был задан при создании буфера. Если каждый раз при воспроизведении вы хотите слушать звук с начала, можно использовать следующий код:

```
// g_pDSBuffer = инициализированный звуковой буфер  
//                с загруженными данными  
if(SUCCEEDED(g_pDSBuffer->SetCurrentPosition(0))) {  
    if(FAILED(g_pDSBuffer->Play(0,0,0))) {  
        // Произошла ошибка  
    }  
}
```

Чтобы остановить воспроизведение просто используйте функцию **IDirectSoundBuffer8::Stop**:

```
if(FAILED(g_pDSBuffer->Stop())) {  
    // Произошла ошибка  
}
```

## Изменение громкости, позиционирования и частоты

Если при создании звукового буфера были указаны соответствующие флаги, вы можете даже во время воспроизведения менять громкость, позиционирование и частоту звукового буфера! Это означает добавление некоторых великолепных функций к вашей звуковой системе, но не впадайте в излишества — эти возможности увеличивают нагрузку на систему. Исключайте флаги неиспользуемых возможностей при создании буферов.

## Регулировка громкости

Работа с громкостью сперва кажется несколько странной. DirectSound воспроизводит звуки с той громкостью, с которой они были записаны. Он не усиливает звуки, чтобы сделать их громче, поскольку это задача аппаратуры.

DirectSound только делает звук тише. Это выполняется путем задания уровня звука в сотых долях децибел в диапазоне от 0 (полная громкость) до –10 000 (тишина). Проблема в том, что громкость звука может понизиться до полной тишины где-нибудь еще, в зависимости от звуковой системы пользователя.

Для изменения громкости вам достаточно вызвать следующую функцию:

```
HRESULT IDirectSoundBuffer8::SetVolume(LONG lVolume);
```

Ее единственный аргумент — это уровень громкости в сотых долях децибел, как я уже говорил. В качестве примера настройки громкости взгляните на приведенный ниже код, который уменьшит громкость на 25 децибел:

```
// g_pDSBuffer = ранее инициализированный звуковой буфер
if(FAILED(g_pDSBuffer->SetVolume(-2500))) {
    // Произошла ошибка
}
```

## Позиционирование

Далее идет *позиционирование (panning)* — это возможность передвигать центр воспроизведения между правым и левым динамиками (как показано на рис. 4.8). Думайте о позиционировании как о регулировке баланса в вашей стереосистеме. Позиционирование определяет, насколько звук будет смещен влево или вправо. Самой дальней левой позиции (работает только левый динамик) соответствует значение –10 000, а самой дальней правой (работает только правый динамик) соответствует 10 000. Все промежуточные значения соответствуют определенному балансу между правым и левым каналами.



**Рис. 4.8.** Обычно динамики воспроизводят звук с одинаковой громкостью (измеряемой в децибелах, или, для краткости, дБ). Позиционирование уменьшает громкость в одном из динамиков и увеличивает в другом, что приводит к возникновению псевдо-трехмерных эффектов

---

<b>ПРИМЕЧАНИЕ</b>	DirectSound определяет два макроса, представляющих крайний левый и крайний правый уровни позиционирования; это <b>DSBPAN_LEFT</b> и <b>DSBPAN_RIGHT</b> соответственно.
-------------------	---

---

А вот волшебная функция:

```
HRESULT IDirectSoundBuffer8::SetPan(LONG lPan);
```

Просто укажите в аргументе **lPan** требуемый уровень позиционирования. Попробуйте в качестве примера установить значение позиционирования буфера равным **-5000**, что уменьшит громкость правого динамика на 50 дБ:

```
// g_pDSBuffer = ранее инициализированный звуковой буфер
if(FAILED(g_pDSBuffer->SetPan(-5000))) {
    // Произошла ошибка
}
```

---

<b>ПРИМЕЧАНИЕ</b>	Чтобы во время воспроизведения можно было управлять позиционированием звука, при создании звукового буфера необходимо указать флаг <b>DSBCAPS_CTRLPAN</b> .
-------------------	---

---

---

<b>ВНИМАНИЕ!</b>	Убедитесь, что первичный звуковой буфер поддерживает 16-разрядный формат, иначе позиционирование может выполняться неточно.
------------------	---

---

## ***Изменение частоты***

Изменение частоты воспроизведения звукового буфера меняет высоту звука. Представьте, что вы можете превратить голос героя игры в верещание белки слегка изменив частоту! Вы можете использовать ту же запись мужского голоса для имитации женского немного повысив его частоту. Вы не верите? Проверьте сами.

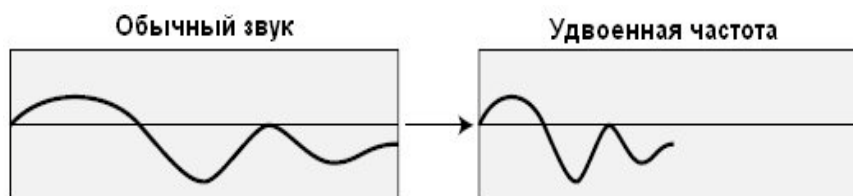
Частота устанавливается следующей функцией:

```
HRESULT IDirectSoundBuffer8::SetFrequency(DWORD dwFrequency);
```

Вам необходимо только задать в аргументе **dwFrequency** желаемое значение, например:

```
// g_pDSBuffer = ранее инициализированный звуковой буфер
if(FAILED(g_pDSBuffer->SetFrequency(22050))) {
    // Произошла ошибка
}
```

Проницательные читатели заметят, что изменение частоты воспроизведения вызывает эффект сжатия звуковой волны из-за чего она воспроизводится за меньшее время, как показано на рис. 4.9.



*Рис. 4.9. Из-за удвоения частоты звук воспроизводится вдвое быстрее и с увеличенной высотой*

## Потеря фокуса

Время от времени другие приложения могут похитить у вас ресурсы, оставляя вас с устройством, состояние которого было изменено. Это обычная ситуация для звуковых буферов, так что вам надо восстанавливать эти потерянные ресурсы, вызывая функцию **IDirectSoundBuffer8::Restore** (у которой нет параметров). Например, если у вас есть потерянный буфер, вы можете восстановить его (и всю связанную с буфером память) используя следующий код:

```
// g_pDSBuffer = ранее инициализированный звуковой буфер,
//              который был потерян
if(FAILED(g_pDSBuffer->Restore())) {
    // Произошла ошибка
}
```

Самое худший эффект потери ресурсов буфера заключается в потере звуковых данных и необходимости их повторной загрузки.

### СОВЕТ

При создании звукового буфера используйте флаг **DSBCAPS\_LOCKSOFTWARE**, говорящий DirectSound о необходимости использовать системную память. Это позволит не беспокоиться о потере ресурсов.

### ПРИМЕЧАНИЕ

Беспокойтесь о падении производительности при отсутствии аппаратной обработки? Система без проблем может работать с несколькими звуковыми буферами. Достаточно попытаться упростить работу системы, задав одинаковый формат воспроизведения для всех звуковых буферов (первичного и вторичных).

## Использование уведомлений

Как вы уже читали, уведомления — это маркеры в звуковом буфере при достижении которых генерируется сигнал о произошедшем событии. Работая с уведомлениями вы получаете возможность узнать, что воспроизведение звука завершено или приостановлено. Вы будете использовать эти уведомления для потокового воспроизведения больших звуков.

Уведомления используют объект с именем **IDirectSoundNotify8**. Его единственная цель — отмечать позиции в звуковом буфере и

генерировать события для приложения, которые могут обрабатываться в цикле обработки сообщений или в отдельном потоке.

Позиции определяются по их смещению в буфере (как показано на рис. 4.10) или с помощью макроса, обозначающего остановку или завершение воспроизведения. Этот макрос определен в DirectSound как **DSBPN\_OFFSETSTOP**.



*Рис. 4.10. Уведомления могут быть размещены (путем указания смещения) в любом месте внутри звукового буфера*

#### **ВНИМАНИЕ!**

Вы не можете указать произвольное смещение в буфере; оно должно быть выровнено по размеру блока выборки. Также уведомления должны быть упорядочены от меньших смещений к большим и никогда два уведомления не могут использовать одно и то же смещение. Если вы используете макрос **DSBPN\_OFFSETSTOP**, это уведомление должно устанавливаться последним. Например, если размер блока равен 2 (монофонический звук, 16 бит) и вы попытаетесь установить смещения 4 и 5, то произойдет сбой, потому что смещения 4 и 5 соответствуют одной выборке.

Чтобы получить объект **IDirectSoundNotify8** вы запрашиваете его через объект **IDirectSoundBuffer8**:

```
// g_pDSBuffer = ранее инициализированный вторичный звуковой буфер
IDirectSoundNotify8 *g_pDSNotify;
if (FAILED(g_pDSBuffer->QueryInterface(IID_IDirectSoundNotify8,
                                        (void**)&g_pDSNotify))) {
    // Произошла ошибка
}
```

#### **ПРИМЕЧАНИЕ**

Чтобы использовать уведомления, вы должны при создании звукового буфера указать флаг **DSBCAPS\_CTRLPOSITIONNOTIFY**.

У интерфейса уведомления есть только одна функция:

---

```
HRESULT IDirectSoundNotify8::SetNotificationPositions(
    DWORD dwPositionNotifies, // Количество уведомлений
    LPCDSBPOSITIONNOTIFY pcPositionNotifies); // Массив смещений
```

---

**ВНИМАНИЕ!**

Вы не можете вызывать функцию `SetNotificationPositions` для буфера, который в данный момент воспроизводится. Если вы хотите изменить позицию уведомления, убедитесь сперва, что воспроизведение буфера установлено. Используйте данную функцию только для вторичных звуковых буферов.

---

Параметр `pcPositionNotifies` является указателем на массив структур `DSBPOSITIONNOTIFY`. Взгляните, что представляет собой эта структура и какую информацию содержит:

```
typedef struct {
    DWORD dwOffset; // Смещение или макрос DSBPN_OFFSET
    HANDLE hEventNotify; // Дескриптор события
} DSBPOSITIONNOTIFY, *LPCDSBPOSITIONNOTIFY;
```

Ловушка здесь — это использование дескриптора события. У события есть два состояния — произошло (установлено) или не произошло (сброшено). Создавая событие вы объявляете переменную дескриптора и инициализируете ее следующим образом:

```
HANDLE hEvent;
hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
```

---

**ПРИМЕЧАНИЕ**

Когда вы закончите использовать событие, следует вызвать `CloseHandle(hEvent)` для освобождения ресурсов.

---

Вы можете создавать столько событий, сколько хотите; важно чтобы в дальнейшем вы могли различать их. Обычно вы создаете одно событие на звуковой канал, но это не строгое правило. Все зависит от вас, и вам решать как вы будете различать, что значит каждое из событий.

---

**СОВЕТ**

Попытайтесь хранить все события в массиве; позднее вы поймете важность их организации подобным образом.

---

Я хочу сделать паузу и показать вам, как установить событие и смещение уведомления. Я буду использовать звуковой буфер размером 65 536 байт и создам два события, представляющие середину и конец буфера соответственно:

```
// g_pDSBNotify = ранее инициализированный объект уведомления
HANDLE g_hEvents[2]; // Глобальный дескриптор
DSBPOSITIONNOTIFY dspn[2]; // Два локальных смещения
// для установки

g_hEvents[0] = CreateEvent(NULL, FALSE, FALSE, NULL);
g_hEvents[1] = CreateEvent(NULL, FALSE, FALSE, NULL);
```

```
dspn[0].dwOffset      = 32768;                // Маркер середины
dspn[0].hEventNotify  = g_hEvents[0];
dspn[1].dwOffset      = DSBPN_OFFSETSTOP;    // Маркер конца звука
dspn[1].hEventNotify  = g_hEvents[1];

if(FAILED(g_pDSBNotify->SetNotificationPositions(2, dspn))) {
    // Произошла ошибка
}
```

Теперь буфер готов, можно двигаться дальше, запустить воспроизведение звукового буфера и позволить событиям сыпаться. Далее следует просто поглядывать на события, ожидая их сигнала. Ожидание событий — это работа функции **WaitForMultipleObjects**:

```
DWORD WaitForMultipleObjects(
    DWORD          nCount,                // Количество ожидаемых событий
                                           // (не больше 64)
    CONST HANDLE *lpHandles,            // Массив дескрипторов ожидаемых
                                           // событий
    BOOL           fWaitAll,             // FALSE (не ждать все)
    DWORD          dwMilliseconds);     // INFINITE (ждать всегда)
```

Здесь интерес представляют только два аргумента — **nCount**, который хранит количество сканируемых событий, и **lpHandles**, являющийся массивом дескрипторов событий, которые сканирует функция. Возвратившись из этой функции вы получите номер позиции произошедшего события в массиве.

---

**ВНИМАНИЕ!**

Функция **WaitForMultipleObject** может сканировать одновременно только 64 объекта, и вам надо следить, чтобы не превысить этот лимит. Функция может также вернуть значение **WAIT\_FAILED**, указывающее что во время ожидания события произошла ошибка. В таком случае просто перезапустите ожидание, и все будет хорошо.

---

В действительности, чтобы получить из возвращаемого значения номер события требуется дополнительная работа. Надо просто вычесть значение **WAIT\_OBJECT\_0** из возвращенного функцией значения, и вы получите число из диапазона от 0 до количества событий минус единица.

Теперь вы видите, почему надо помещать события в массив. Благодаря этому, вы можете быстро определить какое событие произошло, а вот функция, которая воспроизводит звуковой буфер, и ждет возникновения ранее установленных событий:

```
// Функции передается инициализированный звуковой буфер
// с установленными событиями
// g_Events = ранее инициализированный массив событий
//           из двух элементов: HANDLE g_Events[2];
void PlayItAndWait(IDirectSoundBuffer8 *pDSB)
{
    DWORD RetVal, EventNum;

    // Начинаем воспроизведение звука с начала буфера
```



```

pDSBuffer->SetCurrentPosition(0);
pDSBuffer->Play(0,0,0);

while(1) {
    while((RetVal = WaitForMultipleObjects(2, g_Events,
        FALSE, INFINITE) ) != WAIT_FAILED) {
        EventNum = RetVal - WAIT_OBJECT_0;

        // Проверяем событие завершения звука
        // и прерываем цикл
        if(EventNum == 1)
            break;
    }
}
// Остановка воспроизведения
pDSBuffer->Stop();
}

```

Единственная проблема с приведенным выше примером воспроизведения звука заключается в том, что код должен быть помещен в главный цикл обработки сообщений вашей программы и, таким образом, в нем надо сканировать стандартные сообщения Windows. Это возможно, и именно такой способ предпочитает Microsoft в примерах к DirectX SDK.

Проблема с показанной выше постоянной проверкой состояния звукового буфера состоит в том, что она не вписывается в техники модульного проектирования, которые я предпочитаю использовать. Чтобы система работала лучше, создайте отдельный поток, который будет заниматься показанным выше циклом сканирования сообщений.

## Использование потоков для сканирования событий

Мне кажется, я слышал вздох. Не волнуйтесь, использовать потоки для работы с событиями не так трудно. Если вы читали главу 1, «Подготовка к работе с книгой», то уже убедились, что создать поток легко. Трудность в том, как обращаться с таким типом установки.

В главе 1 я писал, что для закрытия потока он должен внутри себя вызвать функцию **ExitThread**. Но как поток узнает, когда это надо делать, если он занят бесконечным сканированием списка сообщений? Решение — добавить еще одно событие, которое будет сообщать о том, что надо закрыть поток!

Чтобы вручную сгенерировать событие вы используете следующий вызов с дескриптором события:

```
SetEvent(hEvent);
```

Чтобы сбросить событие используйте следующий вызов функции:

```
ResetEvent(hEvent);
```

Давайте снова взглянем на цикл сканирования событий, но теперь добавим функции для воспроизведения звукового буфера, для остановки воспроизведения и поддержку потоков:



```
HANDLE g_Events[2]; // Глобальные события
IDirectSoundNotify8 *g_pDSBNotify; // Глобальный объект уведомления

HANDLE g_hThread; // Дескриптор потока

BOOL g_Active = FALSE; // Флаг активности потока
BOOL g_Playing = FALSE; // Флаг воспроизведения звука

// Передайте ранее инициализированный звуковой буфер
// с возможностью уведомлений и данная функция установит
// уведомления для вас.
void PlaySound(IDirectSoundBuffer8 *pDSBuffer)
{
    DSBPOSITIONNOTIFY dspn[1];
    DWORD ThreadId;

    // Останавливаем звук, если он уже воспроизводится
    if(g_Playing == TRUE)
        StopSound(pDSBuffer);

    // Получаем объект уведомления
    pDSBuffer->QueryInterface(IID_IDirectSoundNotify8,
                             (void**)&g_pDSBNotify);

    // Создаем события и поток
    g_hEvents[0] = CreateEvent(NULL, FALSE, FALSE, NULL);
    g_hEvents[1] = CreateEvent(NULL, FALSE, FALSE, NULL);
    g_hThread = CreateThread(NULL, 0,
        (LPTHREAD_START_ROUTINE)MyThread, NULL, 0, &ThreadId);

    // Устанавливаем позицию уведомления
    dspn[0].dwOffset = DSBPN_OFFSETSTOP;
    dspn[0].hEventNotify = g_hEvents[0];
    g_pDSBNotify->SetNotificationPositions(1, dspn);

    // Начинаем воспроизведение и устанавливаем флаг
    pDSBuffer->SetCurrentPosition(0);
    pDSBuffer->Play(0, 0, 0);
    g_Playing = TRUE;
}

void StopSound(IDirectSoundBuffer8 *pDSBuffer)
{
    pDSBuffer->Stop();
    g_Playing = FALSE;

    // Очищаем событие звукового буфера и
    // сигнализируем о закрытии потока
    while(g_Active == TRUE) {
        ResetEvent(g_Events[0]);
        SetEvent(g_Events[1]);
    }

    // Освобождаем все ресурсы
    g_pDSBNotify->Release();
    CloseHandle(g_hEvents[0]);
    CloseHandle(g_hEvents[1]);
    CloseHandle(g_hThread);
}

DWORD WINAPI MyThread(void *lpParameter)
{
    DWORD RetVal, EventNum;
```

```

g_Active = TRUE;
while(1) {
    while((RetVal = WaitForMultipleObjects(2, g_Events,
                                           FALSE, INFINITE) != WAIT_FAILED) {
        EventNum = RetVal - WAIT_OBJECT_0;

        // Проверяем, надо ли закрывать поток
        if(EventNum == 1)
            ExitThread(0);

        // Звук остановлен - сбросим флаг
        if(EventNum == 1) {
            g_Playing = FALSE;
        }
    }
}
}

```

Вот и все. Вам остается только вызвать функцию **PlaySound**, передав ей звуковой буфер с имеющимся в нем звуком; потом вы ждете пока звук не закончится или вызываете функцию **StopSound** для принудительной остановки. Вы можете освободить ресурсы и закрыть поток, вызвав функцию **StopSound** даже если воспроизведение звука уже завершено.

## Загрузка звука в буфер

Теперь, когда у вас есть доступ к звуковому буферу, где взять звуковые данные? Простейший способ - воспользоваться широко распространенным форматом звуковых файлов от Microsoft, называемым *волновые файлы* (*wave files*), и использующим расширение файла .WAV.

Волновые файлы начинаются с небольшого заголовка, а за ним следуют необработанные звуковые данные, которые могут быть как сжатыми (с ними работать труднее), так и несжатыми (с ними иметь дело проще). В данном разделе вы узнаете как прочитать и проанализировать заголовок файла, как прочитать несжатые звуковые данные и как поместить их в звуковой буфер.

Взгляните на структуру, которую я создал чтобы хранить заголовок волнового файла для последующего использования:

```

typedef struct sWaveHeader {
    char  RiffSig[4];           // 'RIFF'
    long  WaveformChunkSize;    // 8
    char  WaveSig[4];          // 'WAVE'
    char  FormatSig[4];         // 'fmt ' (обратите внимание
                                //      на завершающий пробел)
    long  FormatChunkSize;      // 16
    short FormatTag;            // WAVE_FORMAT_PCM
    short Channels;             // количество каналов
    long  SampleRate;          // частота выборки
    long  BytesPerSec;          // байт на секунду
    short BlockAlign;           // выравнивание блока выборки
    short BitsPerSample;        // бит в выборке
    char  DataSig[4];           // 'data'
    long  DataSize;             // размер волновых данных
} sWaveHeader;

```

**ВНИМАНИЕ!**

Большинство волновых файлов при сохранении используют заголовок, аналогичный показанной структуре **sWaveHeader**, но иногда в этот заголовок вставляются дополнительные блоки, что может вызвать путаницу. Например, перед блоком волновых данных может быть вставлен блок комментария. Постарайтесь считывать волновые файлы, которые содержат только один звук, и все должно быть прекрасно.

---

Для обработки заголовка надо открыть волновой файл и сразу же считать из него данные. Структура будет содержать всю информацию, необходимую для определения формата звука, а также размер звуковых данных для последующего чтения.

---

**ПРИМЕЧАНИЕ**

Вы можете определить, соответствует ли звуковой заголовок реальному звуковому файлу, проверив различные поля сигнатур (\*Sig) в структуре **sWaveHeader**. Посмотрите комментарии, в которых указано что каждое из полей должно содержать, и убедитесь при загрузке заголовка, что значения в полях соответствуют указанным. Если в каком-либо из полей значение не совпадает, то, скорее всего, вы не сможете правильно загрузить звук.

---

Сейчас вы можете создать звуковой буфер, основываясь на прочитанных данных, и дальше работать с ним. Я предпочитаю написать пару функций, которые загружают волновой файл в создаваемый ими вторичный звуковой буфер. Первая функция читает и анализирует заголовок волнового файла, создавая по ходу звуковой буфер; вторая читает звуковые данные и помещает их в звуковой буфер.

Первая функция, **CreateBufferFromWAV**, получает указатель на открытый волновой файл, а также указатель на структуру **sWaveHeader**, которая будет заполнена данными заголовка, полученными из волнового файла. По возвращении из функции **CreateBufferFromWAV** вы получаете указатель на новый созданный объект **IDirectSoundBuffer8**, который готов принять звуковые данные, получаемые вызовом функции **LoadSoundData**. Взгляните на код этих двух функций:

```
// g_pDS = ранее инициализированный объект IDirectSound8
IDirectSoundBuffer8 *CreateBufferFromWAV(FILE *fp,
                                         sWaveHeader *Hdr)
{
    IDirectSoundBuffer *pDSB;
    IDirectSoundBuffer8 *pDSBuffer8;
    DSBUFFERDESC dsbd;
    WAVEFORMATEX wfex;

    // Читаем заголовок с начала файла
    fseek(fp, 0, SEEK_SET);
    fread(Hdr, 1, sizeof(sWaveHeader), fp);

    // Проверяем поля сигнатур,
    // возвращаемся в случае ошибки
```

```

if(memcmp(Hdr->RiffSig, "RIFF", 4) ||
    memcmp(Hdr->WaveSig, "WAVE", 4) ||
    memcmp(Hdr->FormatSig, "fmt ", 4) ||
    memcmp(Hdr->DataSig, "data", 4))
    return NULL;

// Устанавливаем формат воспроизведения
ZeroMemory(&wfex, sizeof(WAVEFORMATEX));
wfex.wFormatTag      = WAVE_FORMAT_PCM;
wfex.nChannels       = Hdr->Channels;
wfex.nSamplesPerSec  = Hdr->SampleRate;
wfex.wBitsPerSample  = Hdr->BitsPerSample;
wfex.nBlockAlign     = wfex.wBitsPerSample / 8 * wfex.nChannels;
wfex.nAvgBytesPerSec = wfex.nSamplesPerSec * wfex.nBlockAlign;

// Создаем звуковой буфер, используя данные заголовка
ZeroMemory(&dsbd, sizeof(DSBUFFERDESC));
dsbd.dwSize = sizeof(DSBUFFERDESC);
dsbd.Flags = DSBCAPS_CTRLVOLUME | DSBCAPS_CTRLPAN |
             DSBCAPS_CTRLFREQUENCY;
dsbd.dwBufferBytes = Hdr->DataSize;
dsbd.lpwfxFormat = &wfex;
if(FAILED(g_pDS->CreateSoundBuffer(&dsbd, &pDSB, NULL)))
    return NULL;

// Получаем новую версию интерфейса
if(FAILED(pDSB->QueryInterface(IID_IDirectSoundBuffer8,
                              (void**)&pDSBuffer))) {
    pDSB->Release();
    return NULL;
}

// Возвращаем интерфейс
return pDSBuffer;
}

BOOL LoadSoundData(IDirectSoundBuffer8 *pDSBuffer,
                  long LockPos, FILE *fp, long Size)
{
    BYTE *Ptr1, *Ptr2;
    DWORD Size1, Size2;

    if(!Size)
        return FALSE;

    // Блокируем звуковой буфер с заданной позиции
    if(FAILED(pDSBuffer->Lock(LockPos, Size,
                            (void**)&Ptr1, &Size1,
                            (void**)&Ptr2, &Size2, 0)))
        return FALSE;

    // Читаем данные
    fread(Ptr1, 1, Size1, fp);
    if(Ptr2 != NULL)
        fread(Ptr2, 1, Size2, fp);

    // Разблокируем буфер
    pDSBuffer->Unlock(Ptr1, Size1, Ptr2, Size2);

    // Возвращаем флаг успеха
    return TRUE;
}

```

А вот пример функции, которая использует функции **CreateBufferFromWAV** и **LoadSoundData** для загрузки волнового файла. После возврата из показанной ниже функции **LoadWAV** вы получаете готовый для работы звуковой буфер:

```
IDirectSoundBuffer8 *LoadWAV(char *Filename)
{
    IDirectSoundBuffer8 *pDSBuffer;
    sWaveHeader Hdr;
    FILE *fp;

    // Открываем исходный файл
    if((fp=fopen(Filename, "rb"))==NULL)
        return NULL;

    // Создаем звуковой буфер
    if((pDSBuffer = CreateBufferFromWAV(fp, &Hdr)) == NULL) {
        fclose(fp);
        return NULL;
    }

    // Читаем данные
    fseek(fp, sizeof(sWaveHeader), SEEK_SET);
    LoadSoundData(pDSBuffer, 0, fp, Hdr.DataSize);

    // Закрываем исходный файл
    fclose(fp);

    // Возвращаем новый звуковой буфер с
    // записанным в него звуком
    return pDSBuffer;
}
```

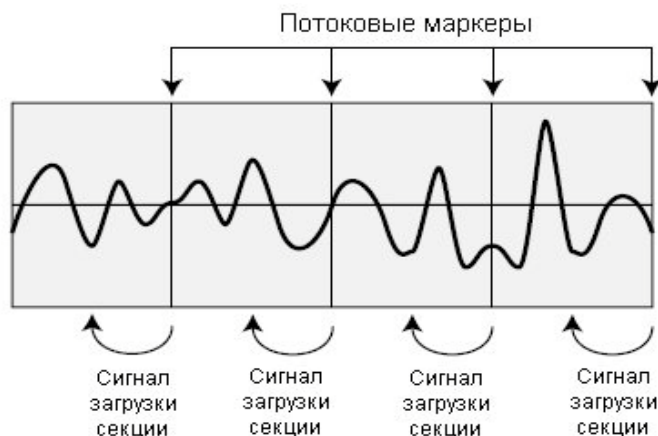
## Потоковое воспроизведение звука

Открою вам маленький секрет — потоковое воспроизведение звука это простой процесс. Секрет заключается в использовании зацикленного воспроизведения, обеспечивающего непрерывный звук без пауз, и постоянную загрузку новых звуковых данных на замену тем, которые уже были воспроизведены.

Секрет трюка заключается в установке нескольких маркеров в звуковом буфере. Когда воспроизводимый звук проходит один из этих маркеров, вы получаете сигнал, что пришло время загружать следующую порцию звуковых данных, вместо воспроизведенной. Таким образом, вы гарантируете, что когда воспроизведение вернется к началу буфера там будут находиться новые звуковые данные. На рис. 4.11 показан звуковой буфер с четырьмя потоковыми маркерами, сообщающими когда должны быть загружены новые звуковые данные.

Для потокового воспроизведения звука вы сначала загружаете звуковыми данными весь буфер (столько данных, сколько поместится в буфер). Запускаете воспроизведение звука и ждете, пока не будет достигнут первый маркер. В этот момент следующий небольшой фрагмент звуковых данных вставляется в только что воспроизведенную секцию.

Воспроизведение продолжается, пока не будет достигнут второй маркер, и в этой точке новые данные загружаются в только что воспроизведенный фрагмент.



*Рис. 4.11. Во вторичном звуковом буфере есть четыре потоковых маркера. Когда воспроизведение достигает одного из этих маркеров, звуковой буфер сигнализирует, что надо загружать новые звуковые данные в только что воспроизведенную секцию*

Процесс продолжается пока весь звук не будет загружен и воспроизведен до того места, где сработает маркер, оповещающий об остановке звука. Если звук зациклен, воспроизведение продолжается путем повторения последовательности действий, о которых вы прочитали.

В предыдущем разделе, «Загрузка звука в буфер», я написал функцию с именем **LoadSoundData**, которая загружает звуковые данные в указанный вами буфер. В потоке, обрабатывающем события уведомлений, вы используете функцию загрузки для поддержания потока данных вместо завершенных, обеспечивая заполнение буфера звуковой информацией.

Вот как это все делается:

1. Создаем звуковой буфер, скажем, размером 65536 байт.
2. Устанавливаем четыре позиции уведомлений (в конце каждой четверти звукового буфера).
3. Загружаем в буфер столько данных, сколько поместится.
4. Запускаем воспроизведение звукового буфера, указав флаг **DSBPLAY\_LOOP**.
5. При получении уведомления о событии загружаем в только что воспроизведенную секцию новые данные. Продолжаем, пока не достигнем уведомления, после которого не останется данных для загрузки и тогда воспроизводим оставшуюся часть звука.
6. Чтобы определить, какое событие отмечает конец звука, определите модуль размера звука по размеру буфера (остаток от деления размера звука на размер буфера). Разделите значение модуля на четыре (количество уведомлений) чтобы определить, какое из событий используется для оповещения конечной позиции звука.

Демонстрационная программа Stream анализирует заголовок волнового файла, создает звуковой буфер, устанавливает четыре уведомления и события, а затем воспроизводит звук, буферизуя данные по мере возникновения событий. Этот процесс продолжается, пока не будет достигнут конец звука, и воспроизведение завершается.

Код демонстрационной программы Stream вы найдете в главе 6, «Создаем ядро игры». Там вы обнаружите набор рабочих функций, которые можно использовать для потокового воспроизведения звука в собственном проекте.

---

**ПРИМЕЧАНИЕ**

Весь код примера потокового воспроизведения (называемого Stream) находится на прилагаемом к книге CD-ROM (загляните в папку BookCode\Chap04\Stream). Из-за его длины я не могу привести весь код здесь, но он следует представленной в данном разделе методике. Для потокового воспроизведения звука вызовите функцию `PlayStreamedSound` с именем звукового файла, который вы хотите воспроизвести, и функция позаботится за вас обо всем остальном.

---

## Работа с DirectMusic

В то время, как DirectSound обрабатывает цифровые звуки, DirectMusic обрабатывает воспроизведение музыки из файлов MIDI (файлы с расширением .MID), родных файлов DirectMusic (файлы .SGT) и цифровых записей песен, хранящихся в волновом формате (файлы .WAV). Какой из этих форматов лучше для вас — вот вопрос, требующий ответа.

У каждого есть преимущества и недостатки. Настоящая магия DirectMusic проявляется, когда используют его родной формат. Песня в родном формате DirectMusic может состоять из небольших музыкальных партитур, которые могут случайно воспроизводиться одна за другой в различных аккордах. Случайный выбор партитур и аккордов означает, что музыка никогда не будет той же самой — она все время будет меняться. Добавьте изменение темпа, и вы получите крутую музыкальную систему.

---

**ПРИМЕЧАНИЕ**

Другая возможность — использование *мотивов (motif)*, то есть звуков, накладывающихся на музыкальные сегменты при их воспроизведении. Они служат оттенками, смешивающимися с песней. Например, если игрок достиг цели, чтобы показать это можно воспроизвести короткий звук рожка.

---

Преимущество использования файлов MIDI — их широкая поддержка. В Интернете вы найдете тысячи песен в этом формате, и сотни программных пакетов для создания MIDI-музыки. Замечательно, что вы можете использовать цифровые записи звуков в качестве инструментов.

Представьте себе — вместо обычных инструментов вы можете использовать другие записанные в цифровой форме звуки, такие как

выстрелы оружия, вопли обезьян, или что-угодно еще, поражающее воображение. Вы можете наконец записать ту новую мелодию, которую хотели услышать в вашей собственной игре и использовать для этого музыкальный формат MIDI.

Использование цифровых записей инструментов гарантирует, что ваши песни будут одинаково звучать на всех компьютерах. Вы достигаете этого, используя DirectSound для синтеза звуковых эффектов. DirectMusic имеет возможность создавать собственные интерфейсы DirectSound или использовать уже созданные.

---

**ПРИМЕЧАНИЕ** Путь, которым музыкальные данные поступают в синтезатор (DirectSound) называются *аудио-путем (audio path)* и вы можете перехватить музыку и воспроизвести точно также, как обычный звуковой буфер DirectSound. Только вообразите - захват звуковой дорожки и создание на ее основе трехмерного звукового буфера! Это все возможно, и это делает DirectMusic таким привлекательным.

---

Использование MIDI или родного формата DirectMusic имеет одно общее преимущество (возможность менять темп воспроизведения), добавляющее одну замечательную возможность — ускорять и замедлять музыку, согласно происходящим на экране действиям. Когда на экране разворачиваются активные действия, увеличьте темп музыки, а затем уменьшите его, когда обстановка станет более спокойной.

---

**ПРИМЕЧАНИЕ** Проблема с родным форматом состоит в том, что вам потребуется время на привыкание к написанию партитур и аккордов. Нет никакой возможности показать вам основы в ограниченном объеме книги, так что я отсылаю вас к DirectMusic Producer — музыкальному редактору от Microsoft. Вы найдете его на прилагаемом к книге CD-ROM или можете загрузить с сайта <http://www.microsoft.com>.

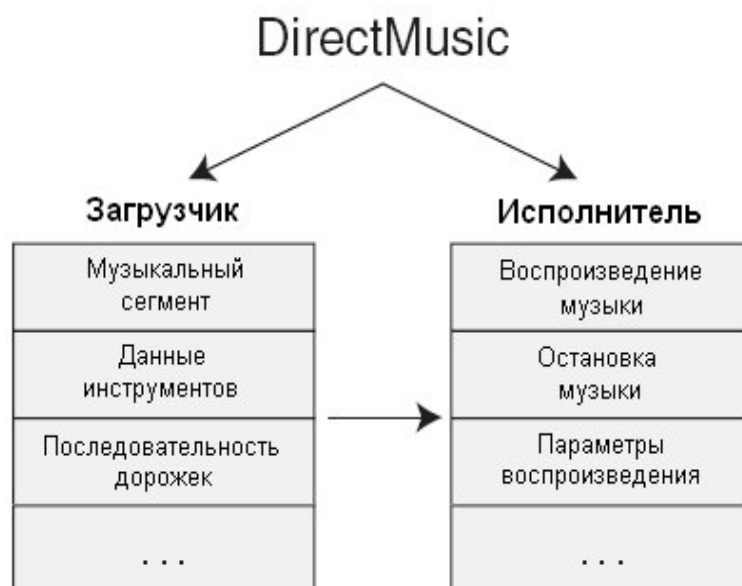
---

Цифровые записи музыки создают бесподобную систему... в каком-то роде. Хотя качество превосходно, вы не можете менять песню в соответствии с действиями. Песня звучит точно так, как записана, ничего больше но и ничего меньше.

## Начинаем работу с DirectMusic

Теперь, когда вы заинтересовались, пойдем дальше. Первый этап использования DirectMusic — это создание главного объекта, называемого *исполнителем (performance)*, который представляет музыкальную систему. Затем, создается объект, называемый *загрузчик (loader)*, который загружает все основные музыкальные файлы. Взаимодействие между этими двумя объектами показано на рис. 4.12.





*Рис. 4.12. Объект загрузчика предоставляет данные, необходимые объекту исполнителя для воспроизведения музыки*

Затем вы загружаете реальные музыкальные сегменты в объекты *сегментов* (*segment*). Для создания длинных или более динамичных песен вы можете загрузить несколько сегментов и воспроизводить их один за другим. В этой главе я буду иметь дело только с одним сегментом (который представляет всю песню целиком).

В DirectMusic нет функции помогающей вам создать или инициализировать основной интерфейс DirectMusic, поэтому вам надо самостоятельно инициализировать COM-систему. Для инициализации COM вызовите следующую функцию:

```
CoInitialize(NULL);
```

Делайте это один раз, когда начинаете использование DirectMusic, поскольку при этом сохраняется внутренний счетчик количества инициализаций. Каждому вызову этой функции должен соответствовать вызов для завершения работы COM:

```
CoUninitialize();
```

Он уменьшает счетчик ссылок использующих COM и, когда значение станет равным 0, COM-система будет выгружена из памяти. Это повышает эффективность использования памяти и все COM-объекты следуют этой процедуре. Чтобы создать объект вы используете функцию **CoCreateInstance**. В следующих двух разделах я покажу вам, как создать объекты исполнителя и загрузчика соответственно.

## Создание объекта исполнителя

Объект исполнителя — это большая шишка и основной объект, с которым вы будете работать. У вас может быть несколько объектов исполнителей, но я рекомендую использовать только один. Для создания объекта исполнителя

сперва объявите экземпляр объекта **IDirectMusicPerformance8**, а затем вызовите функцию **CoCreateInstance**, как показано ниже:

```
// Глобальный объект исполнителя
IDirectMusicPerformance8 *g_pDMPPerformance;

CoCreateInstance(CLSID_DirectMusicPerformance, NULL,
    CLSCTX_INPROC, IID_IDirectMusicPerformance8,
    (void**) &g_pDMPPerformance);
```

---

**ПРИМЕЧАНИЕ**      Функция **CoCreateInstance** возвращает значение **S\_OK**, если вызов завершен успешно. Любое другое значение свидетельствует об ошибке.

---

Объект исполнителя необходимо инициализировать. При этом создаются объекты **DirectMusic** и **DirectSound**, они создают звуковые буферы и устанавливают возможности воспроизведения. Также устанавливается аудио-путь по умолчанию, по которому воспроизводится музыка. Стандартные параметры предполагают использование 128 каналов (инструментов) и включение стереофонии и эффекта эха (отражения звука от объектов). Вот функция, вызов которой делает все это:

```
HRESULT IDirectMusicPerformance8::InitAudio(
    IDirectMusic **ppDirectMusic,    // NULL
    IDirectSound **ppDirectSound,    // NULL
    HWND          hWnd,              // Дескриптор родительского окна
    DWORD          dwDefaultPathType, // Тип аудио-пути по умолчанию
                                         // используйте DMUS_APATH_SHA-
                                         // RED_STEREOPLUSREVERB
    DWORD          dwPChannelCount,  // Количество каналов
                                         // используйте 128
    DWORD          dwFlags,           // DMUS_AUDIOF_ALL (разрешить все
                                         // музыкальные возможности)
    DMUS_AUDIOPARAMS *pParams);      // NULL (структура параметров)
```

Здесь много информации, но почти все сказано в комментариях. Вам не нужны указатели на внутренние объекты **DirectMusic** и **DirectSound**, поэтому пропустим их. Вы должны дать функции дескриптор родительского окна — это необходимо. Для других параметров можно оставить значения, приведенные в комментариях прототипа функции **InitAudio**.

Вот пример вызова функции:

```
// g_pDMPPerformance = ранее созданный объект исполнителя
if (FAILED(g_pDMPPerformance->InitAudio(NULL, NULL, hWnd,
    DMUS_APATH_SHARED_STEREOPLUSREVERB, 128,
    DMUS_AUDIOF_ALL, NULL))) {
    // Произошла ошибка
}
```

## Создание объекта загрузчика

Следующий этап использования **DirectMusic** — создание объекта загрузчика. Объект в основном является системой кэширования, ускоряющей загрузку

данных и выполняющей загрузку необходимых песням файлов (таких, как цифровые записи, используемые для инструментов).

Загрузчик представляет объект **IDirectMusicLoader8**. Вы создаете его с помощью следующего кода:

```
IDirectMusicLoader8 *g_pDMLoader; // Глобальный объект загрузчика

CoCreateInstance(CLSID_DirectMusicLoader, NULL,
                CLSCTX_INPROC, IID_IDirectMusicLoader8,
                (void**) &g_pDMLoader);
```

---

**ВНИМАНИЕ!**

Убедитесь, что в вашем приложении вы создаете только один объект **IDirectMusicLoader8**. Это помогает кэшированию и управлению часто используемыми данными и ресурсами, необходимыми при работе с DirectMusic.

---

Следующий этап использования загрузчика — указание ему, в каком каталоге следует искать файлы. На этот каталог ссылаются как на *каталог поиска по умолчанию (default search directory)*. Обычно при загрузке единственного музыкального файла, такого как MIDI-файл, установка каталога по умолчанию не требуется, поскольку вы просто сообщаете загрузчику полный путь. Но для файлов родного формата DirectMusic объект загрузчика должен знать, где искать вспомогательные файлы.

Установка каталога поиска по умолчанию — это работа функции **IDirectMusicLoader8::SetSearchDirectory**:

```
HRESULT IDirectMusicLoader8::SetSearchDirectory(
    REFGUID rguidClass, // Класс (GUID_DirectMusicalTypes)
    WCHAR *pwszPath,   // Путь к каталогу (широкие символы)
    BOOL fClear);      // FALSE - очистить кэш
```

Показанному вызову реально нужен только один параметр — устанавливаемый путь к каталогу для поиска. Будьте внимательны — это должна быть строка широких символов, так что преобразуйте ее или используйте тип данных **WCHAR**.

---

**ПРИМЕЧАНИЕ**

Чтобы объявить строку широких символов используйте следующий код:

```
WCHAR *Text = L"Testing";
```

Чтобы преобразовать строку обычных символов в строку широких символов используйте функцию **mbstowcs**, как показано ниже:

```
char Text[] = "Roleplaying is fun!";           // Буфер
                                                // исходного текста
WCHAR WText[256];                             // Буфер для
                                                // преобразованного текста
// Преобразуем 256 символов из источника в приемник
mbstowcs(WText, Text, 256);
```

---

Для простоты я обычно в качестве каталога поиска задаю текущий каталог приложения. Таким образом вы можете сослаться на файлы песен из

ваших собственных подкаталогов (например, `.\Songs\` — обратите внимание на точку, означающую текущий каталог).

Вот пример установки текущего каталога в качестве каталога поиска по умолчанию:

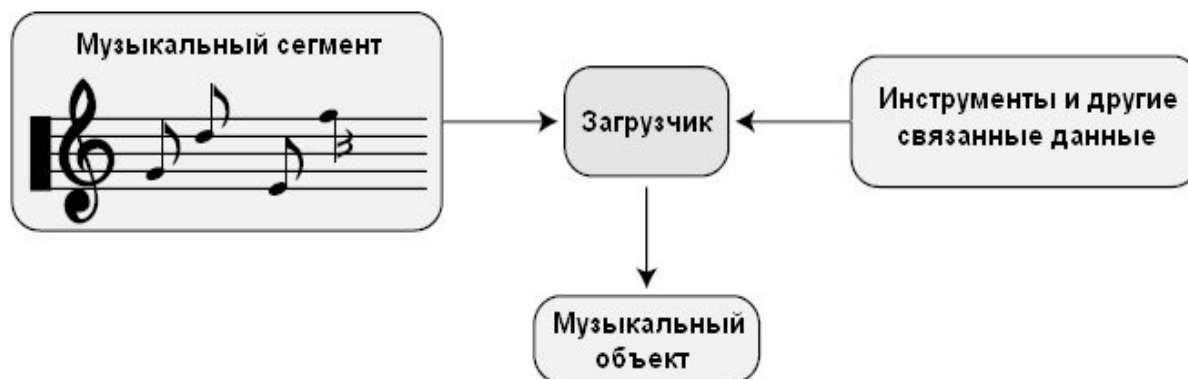
```
// g_pDMLoader = ранее инициализированный объект загрузчика
CHAR strPath[MAX_PATH]; // Текущий путь
WCHAR wstrPath[MAX_PATH]; // Буфер широких символов

GetCurrentDirectory(MAX_PATH, strPath);
mbstowcs(wstrPath, strPath, MAX_PATH);

if(FAILED(g_pDMLoader->SetSearchDirectory(
    GUID_DirectMusicAllTypes, wstrPath, FALSE))) {
    // Произошла ошибка
}
```

## Работа с музыкальными сегментами

Теперь система инициализирована и загрузчик готов — настало время начать загрузку песни и включить ее воспроизведение. Это работа объекта **IDirectMusicSegment8**. Объект загрузчика DirectMusic (как показано на рис. 4.13) загружает музыку и данные инструментов и создает объект **IDirectMusicSegment8** за вас. Учтите, что процесс загрузки состоит из двух этапов — сперва вы загружаете музыкальный сегмент, содержащий ноты для воспроизведения.



*Рис. 4.13. Объект загрузчика отвечает за получение данных, таких как музыкальные партитуры и данные инструментов, необходимые для создания музыкального объекта*

### Загрузка музыкальных сегментов

Первый этап — инициализация объекта структуры описателя с именем **DMUS\_OBJECTDESC**, содержащей информацию о том, что вы загружаете (о песне). Вот эта структура:

```
typedef struct {
    DWORD        dwSize;           // Размер структуры
    DWORD        dwValidData;      // Флаги, определяющие
                                   // действительные поля
    GUID         guidObject;       // Уникальный GUID объекта
    GUID         guidClass;        // CLSID_DirectMusicSegment
}
```

```

FILETIME      ftDate;          // Дата последнего
                                // редактирования объекта
DMUS_VERSION  vVersion;       // Структура, содержащая
                                // информацию о версии
WCHAR  wszName[DMUS_MAX_NAME]; // Имя объекта
WCHAR  wszCategory[DMUS_MAX_CATEGORY]; // Категория объекта
WCHAR  wszFileName[DMUS_MAX_FILENAME]; // Имя файла для загрузки
LONGLONG llMemLength; // Размер данных в памяти
LPBYTE  pbMemData; // Указатель на данные в памяти
IStream *pStream; // Интерфейс потока для загрузки
} DMUS_OBJECTDESC;

```

К счастью, большинство полей в **DMUS\_OBJECTDESC** можно игнорировать. Первое, заслуживающее внимания поле — это **dwValidData**. Оно хранит комбинацию флагов, сообщающих загрузчику, какие поля в структуре используются. Например, если вы хотите использовать **wszFilename** и **guidClass**, установите соответствующие флаги. Список флагов приведен в таблице 4.4.

Таблица 4.4. Флаги **dwValidData**

Флаг	Описание
<b>DMUS_OBJ_CATEGORY</b>	Действительно значение <b>wszCategory</b> .
<b>DMUS_OBJ_CLASS</b>	Действительно значение <b>guidClass</b> .
<b>DMUS_OBJ_DATE</b>	Действительно значение <b>ftDate</b> .
<b>DMUS_OBJ_FILENAME</b>	Действительно значение <b>wszFileName</b> .
<b>DMUS_OBJ_FULLPATH</b>	<b>wszFileName</b> содержит полный путь к объекту.
<b>DMUS_OBJ_LOADED</b>	Объект уже загружен.
<b>DMUS_OBJ_MEMORY</b>	Объект в памяти. Действительны значения <b>llMemLength</b> и <b>pbMemData</b> .
<b>DMUS_OBJ_NAME</b>	Действительно значение <b>wszName</b> .
<b>DMUS_OBJ_OBJECT</b>	Действительно значение <b>guidObject</b> .
<b>DMUS_OBJ_STREAM</b>	Действительно значение <b>pStream</b> .
<b>DMUS_OBJ_URL</b>	<b>wszFileName</b> представляет адрес URL.
<b>DMUS_OBJ_VERSION</b>	Действительно значение <b>vVersion</b> .

Структура **DMUS\_OBJECTDESC** передается функции **IDirectMusicLoader8::GetObject**, обеспечивающей загрузку и размещение в объекте сегмента всех связанных файлов данных. Вот прототип функции:

```

HRESULT IDirectMusicLoader8::GetObject(
    LPDMUS_OBJECTDESC pDesc, // Указатель на структуру DMUS_OBJECTDESC
    REFIID riid,             // IID_IDirectMusicSegment8
    LPVOID FAR *ppv);        // Указатель на новый загруженный объект

```

Вы говорите функции **GetObject** использовать структуру, которую вы инициализировали и загрузить данные в объект **IDirectMusicSegment8**. Конечно, вам сперва нужно объявить объект сегмента, чтобы иметь дело с функцией, которая обрабатывает загрузку фрагмента музыки и возвращает объект сегмента (или **NULL**, если произошла ошибка).

```
// g_pDMLoader = ранее инициализированный объект загрузчика
//               с установленным каталогом поиска
IDirectMusicSegment8 *LoadSong(char *Filename)
{
    DMUS_OBJECTDESC dmod;
    IDirectMusicSegment8 *pDMSegment;

    ZeroMemory(&dmod, sizeof(DMUS_OBJECTDESC));

    dmod.dwSize      = sizeof(DMUS_OBJECTDESC);
    dmod.guidClass   = CLSID_DirectMusicSegment;
    dmod.dwValidData = DMUS_OBJ_CLASS | DMUS_OBJ_FILENAME |
                      DMUS_OBJ_FULLPATH;
    mbstowcs(dmod.wszFileName, Filename, MAX_PATH);

    if(FAILED(g_pDMLoader->GetObject(&dmod,
                                     IID_IDirectMusicSegment8, (LPVOID)&pDMSegment)))
        return NULL;

    // Загрузка завершена
    return p_DMSegment;
}
```

## Загрузка инструментов

Загрузчик DirectMusic устанавливает используемые по умолчанию инструменты, когда вы используете песни в родном формате MIDI, но что делать в тех случаях, когда вы хотите изменить параметры инструментов MIDI? Тогда вам потребуется ваш друг DirectMusic, позволяющий использовать ваши собственные данные инструментов.

Инструменты называются *модификаторами (patches)*, а набор модификаторов называется *DLS-данными инструментов (DLS instrument data — от Downloadable Sounds)*, которые включаются в *коллекции инструментов (instrument collections)*. Модификаторы нумеруются последовательностью из трех значений: *старшего значащего байта (most significant byte, MSB)*, *младшего значащего байта (least significant byte, LSB)* и *номера модификатора (patch number)*.

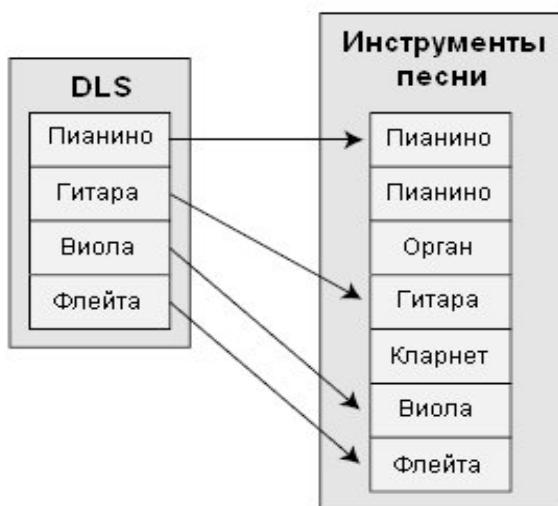
Основные модификаторы MIDI стандартизованы, так что модификатор с номером 1 (пианино) всегда будет относиться к пианино. Если вы хотите использовать новый модификатор пианино, достаточно просто загрузить его из DLS-коллекции. DirectMusic поставляется с коллекцией инструментов, соответствующей спецификации General MIDI, называемой *набор GM/GS (GM/GS set)* созданной Roland.

**ПРИМЕЧАНИЕ**

Чтобы создать собственный DLS установите DirectMusic Producer, находящийся на CD-ROM. В меню выберите **File, New** и создайте DLS-коллекцию (**DLS Collection**). Добавьте в список волновые файлы, а затем добавьте инструменты, убедившись, что им назначены соответствующие волновые данные.

Если вы делаете новые инструменты для замены модификаторов General MIDI, убедитесь, что у них MSB и LSB равны 0; в ином случае `bcgikmpreqnt` различные значения для каждого инструмента, чтобы гарантировать отсутствие вторжений в пространство того или другого инструмента. Если вам потребуется дополнительная помощь, обратитесь к файлу справки DirectMusic Producer.

Если вы хотите использовать только пару новых инструментов, любым способом сохраните их в DLS. Загружая новый DLS, вы перезаписываете ранее загруженные в память инструменты (как показано на рис. 4.14). После того, как DLS-коллекция готова к использованию сообщите DirectMusic, что эту коллекцию надо использовать для ваших музыкальных сегментов.



**Рис. 4.14.** Загружая DLS-инструменты вы либо перезаписываете данные существующего инструмента, либо вставляете данные нового инструмента туда, где никаких инструментов нет

Для загрузки DLS-коллекции вам необходимо получить объект **IDirectMusicCollection8** через объект загрузчика. Вы снова используете функцию **IDirectMusicLoader8::GetObject**, но на этот раз указываете объект коллекции и имя файла. Вот функция, которая загружает для вас DLS-коллекцию и возвращает указатель на объект загруженной коллекции для дальнейшей работы:

```
IDirectMusicCollection8 *LoadDLSCollection(char *Filename)
{
    DMUS_OBJECTDESC dmod;
    IDirectMusicCollection8 *pDMCollection;

    ZeroMemory(&dmod, sizeof(DMUS_OBJECTDESC));
    dmod.dwSize = sizeof(DMUS_OBJECTDESC);
    dmod.guidClass = CLSID_DirectMusicCollection;
    dmod.dwValidData = DMUS_OBJ_CLASS | DMUS_OBJ_FILENAME |
                      DMUS_OBJ_FULLPATH;
```

```

    mbstowcs(dmod.wszFileName, Filename, MAX_PATH);
    if(FAILED(g_pDMLoader->GetObject(&dmod,
        IID_IDirectMusicCollection8, (void**)pDMCollection)))
        return NULL;

    // Возвращаем указатель на объект коллекции
    return IDirectMusicCollection8;
}

```

Теперь коллекция загружена, но вам надо еще назначить ее сегменту. Это делается путем установки параметра сегмента трека с помощью функции **IDirectMusicSegment8::SetParam**:

```

HRESULT IDirectMusicSegment8::SetParam(
    REFGUID rguidType, // GUID устанавливаемого параметра
    DWORD dwGroupBits, // На какой трек оказывает эффект (0xFFFFFFFF)
    DWORD dwIndex,     // 0
    MUSIC_TIME mtTime, // Когда применять изменения - используйте 0
    void* pParam);     // Новый параметр или NULL если не требуется

```

Сейчас вы хотите установить параметр, задающий тип DLS-коллекции, которому соответствует значение GUID **GUID\_ConnectToDLSCollection**. Вы хотите, чтобы параметр влиял на каждый трек и изменения вступили в силу немедленно. Для этого используйте следующий фрагмент кода (который загружает DLS и устанавливает ее для ранее загруженного сегмента):

```

IDirectMusicCollection8 *pDMCollection;

if((pDMCollection = LoadDLSCollection("MyDLS.dls")) != NULL)
    pDMSegment->SetParam(GUID_ConnectToDLSCollection,
        0xFFFFFFFF, 0, 0, (void*)pDMCollection);

```

Иногда вам надо будет использовать предлагаемую по умолчанию коллекцию, что достигается вызовом **GetObject** со значением GUID **GUID\_DefaultGMCollection** в поле объекта класса:

```

IDirectMusicCollection8 *GetDefaultCollection()
{
    DMUS_OBJECTDESC mod;
    IDirectMusicCollection8 *pDMCollection;

    ZeroMemory(&dmod, sizeof(DMUS_OBJECTDESC));
    dmod.dwSize = sizeof(DMUS_OBJECTDESC);
    dmod.guidObject = GUID_DefaultDMCollection;
    dmod.dwValidData = DMUS_OBJ_OBJECT;

    if(FAILED(g_pDMLoader->GetObject(&dmod,
        IID_IDirectMusicCollection8,
        (void**)pDMCollection)))
        return NULL;

    return pDMCollection;
}

```



Вызов показанной выше функции **GetDefaultCollection** создаст объект коллекции инструментов, содержащий DLS-данные инструментов по умолчанию, которые вы можете использовать.

## Настройки для MIDI

Вся песня в памяти (с инструментами) и почти готова к использованию. Осталось лишь пара проблем, которыми следует заняться. Во-первых, поскольку система должна подготовить себя в соответствии со спецификацией General MIDI, вам необходимо сообщить системе является ли загружаемый файл MIDI-файлом.

Чтобы сообщить DirectMusic, что сегмент является MIDI-файлом вы снова устанавливаете параметры сегмента трека функцией **IDirectMusicSegment8::SetParam**. На этот раз вы используете значение GUID **GUID\_StandardMidiFile**:

```
pDMSegment->SetParam(GUID_StandardMidiFile,
                    0xFFFFFFFF, 0, 0, NULL);
```

Вы можете поместить показанный ниже вызов функции **SetParam** в функцию **LoadSongFile** (после того, как песня полностью загружена):

```
if (FAILED(g_pDMLoader->GetObject(&dmod,
    IID_IDirectMusicSegment8, (LPVOID)&pDMSegment)))
    return NULL;

// Загрузка завершена
// Устанавливаем признак MIDI-файла
if (strstr(Filename, ".mid") != NULL)
    pDMSegment->SetParam(GUID_StandardMidiFile,
        0xFFFFFFFF, 0, 0, NULL);

return p_DMSegment;
}
```

## Установка инструментов

Следующий этап подготовки сегмента к воспроизведению — установка данных инструментов путем их загрузки в объект исполнителя. Это выполняется с помощью вызова **IDirectMusicSegment8::Download**:

```
HRESULT IDirectMusicSegment8::Download(IUnknown *pAudioPath);
```

---

**ВНИМАНИЕ!**

Выполняйте этот вызов только для MIDI-файлов, поскольку он меняет способ восприятия музыкальной информации. Если вы поэкспериментируете с ним, то увидите, что отдельные данные трека изменяются или теряются.

---

Единственный параметр функции — аудио-путь для которого загружаются данные инструментов. В данном случае это объект исполнителя, так что используйте следующий код:

```
if(FAILED(g_pDMSegment->Download(g_pDMPerformance))) {
    // Произошла ошибка
}
```

**СОВЕТ**

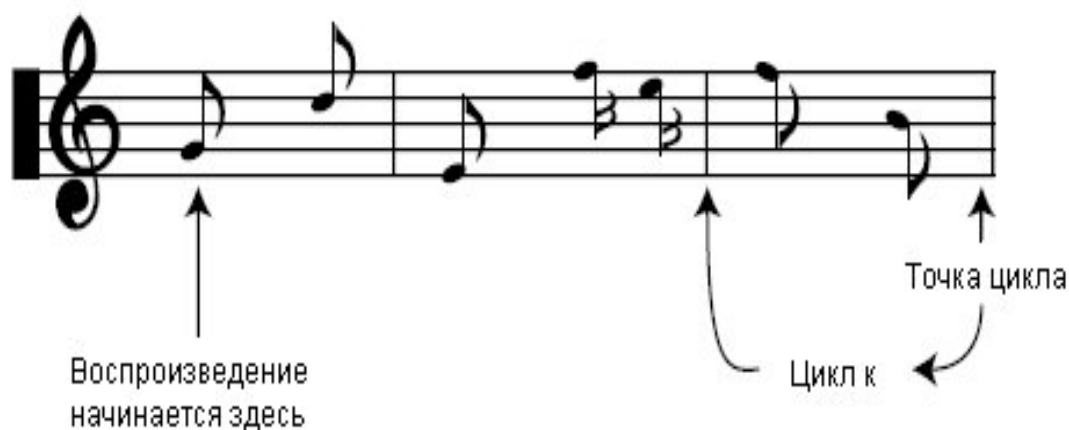
Чтобы изменить инструменты (например, назначить DLS сегменту) вы сперва должны выгрузить данные инструментов. После выгрузки инструментов вы можете загрузить новые данные инструментов и продолжить воспроизведение песни.

Когда вы завершите работу с музыкальным сегментом, необходимо выполнить вызов **IDirectMusicSegment8::Unload**, освобождающий данные инструментов. Делайте это после того, как остановили воспроизведение сегмента и завершили работать с ним или когда вы переключаете коллекцию инструментов. Вызов идентичен **IDirectMusicSegment8::Download**, так что я пропущу прототип и сразу покажу реальный код:

```
if(FAILED(g_pDMSegment->Unload(g_pDMPerformance))) {
    // Произошла ошибка
}
```

**Использование циклов и повторов**

Последний шаг перед воспроизведением — установка точек повтора и количества повторений цикла. Например, если у вас есть ритм и вы хотите повторить его небольшую часть несколько раз, то устанавливаете начальную и конечную точки цикла (как показано на рис. 4.15), а затем задаете количество повторений цикла.



**Рис. 4.15.** Установите начальную и конечную точки цикла внутри песни, чтобы воспользоваться циклами и возможностями повторов *DirectMusic*

Установка точек цикла — это назначение функции **IDirectMusicSegment8::SetLoopPoints**:

```
HRESULT IDirectMusicSegment8::SetLoopPoints(
    MUSIC_TIME mtStart,
    MUSIC_TIME mtEnd);
```

**ВНИМАНИЕ!**

Обратите внимание на использование **MUSIC\_TIME** — единиц измерения времени, применяемых в DirectMusic. Это измерение времени основано на темпе песни, а не на таймере, так что иногда с ним сложно работать. Выбор времени это еще одна проблема и лучше обратиться к документации DirectX SDK. Для текущих целей, когда вы хотите, чтобы все изменения давали эффект немедленно, безопасно для времени задавать значение 0.

Обычно вы хотите, чтобы песня была воспроизведена вся целиком до завершения, так что нет причин заморачиваться с функцией **SetLoopPoints**. Если вы все же решитесь на это, помните, что измерение времени основано на темпе (за дополнительными сведениями по этой теме обратитесь к документации DX SDK).

После установки точек цикла (даже если у вас их нет), вы готовы указать сколько раз будет повторена песня. Если вы хотите, чтобы песня была воспроизведена один раз и остановлена, число повторов должно быть равно нулю. Если вы хотите чтобы песня была воспроизведена дважды, повторите ее один раз.

Вы устанавливаете количество повторов функцией **IDirectMusicSegment8::SetRepeats**, у которой всего один параметр — количество повторений цикла песни (или макрос **DMUS\_SEG\_REPEAT\_INFINITE**, который вызывает бесконечное воспроизведение песни):

```
pDMSegment->SetRepeats(0); // Воспроизводим песню один раз
                          // (нет циклов)
```

### **Воспроизведение и остановка сегмента**

Теперь, наконец-то пришло время воспроизвести вашу песню. Да, дорога была долгой и трудной, но мы достигли ее конца. Объект исполнителя воспроизводит сегмент с помощью функции **IDirectMusicPerformance8::PlaySegmentEx**:

```
HRESULT IDirectMusicPerformance8::PlaySegmentEx(
    IUnknown *pSource,          // Воспроизводимый сегмент
    WCHAR *pwzSegmentName,     // NULL - не используется
    IUnknown *pTransition,     // Сегмент перехода - используйте NULL
    DWORD dwFlags,             // Флаги изменения поведения
    __int64 i64Starttime,       // Когда начинать воспроизведение
                                // 0 - немедленно
    IDirectMusicSegmentState **ppSegmentState, // Указатель на объект
                                                // получающий объект
                                                // состояния сегмента

    IUnknown *pFrom,           // NULL
    IUnknown *pAudioPath);     // Используемый аудио-путь или
                                // NULL для аудио-пути по умолчанию
```

Ничего себе! Как много всего; к счастью вам не надо использовать все эти параметры. Вы видите, что указатель на сегмент является первым

аргументом, но что такое *состояние сегмента* (*segment state*)? Это объект который отслеживает статус сегмента. Он вам не нужен, так что укажите **NULL**.

Флаги поведения позволяют изменить время начала сегмента, начинается ли он с такта, выравнивается по предшествующему темпу, или как вы указали. Поскольку сейчас нас интересует только воспроизведение песни, можно пропустить флаги и сообщить функции **PlaySegmentEx** о немедленном начале воспроизведения.

Быстро начать воспроизведение сегмента можно с помощью следующего фрагмента кода:

```
if (FAILED(g_pDMPPerformance->PlaySegmentEx(g_pDMSegment,
                                             NULL, NULL, 0, 0, NULL, NULL, NULL))) {
    // Произошла ошибка
}
```

Чтобы остановить воспроизведение сегмента используйте вызов **IDirectMusicPerformance::Stop**:

```
HRESULT IDirectMusicPerformance8::Stop(
    IDirectMusicSegment *pSegment, // Останавливаемый сегмент
    IDirectMusicSegmentState *pSegmentState, // Состояние для остановки
    MUSIC_TIME mtTime, // Время остановки (0 - немедленно)
    DWORD dwFlags); // Поведение времени остановки
```

Снова вызов получает в аргументе сегмент, а также время, когда вы хотите выполнить остановку. Вся перечисленная информация не требуется; достаточно предоставить указатель на объект сегмента и время остановки воспроизведения, как показано ниже:

```
if (FAILED(g_pDMPPerformance->Stop(g_pDMSegment, NULL, 0, 0))) {
    // Произошла ошибка
}
```

## Выгрузка данных сегмента

После того, как вы остановили сегмент и завершили работу с ним, необходимо выгрузить данные инструментов:

```
pDMSegment->Unload(g_pDMPPerformance);
```

Вам также нужно обратиться к загрузчику для освобождения кэшированных данных с помощью вызова **IDirectMusicLoader8::ReleaseObjectByUnknown**:

```
HRESULT IDirectMusicLoader8::ReleaseObjectByUnknown(
    IUnknown *pObject);
```

Метод **ReleaseObjectByUnknown** получает один параметр — указатель на объект выгружаемого сегмента. Когда выгрузка произведена вы можете освободить СОМ-объект сегмента. Вот два вызова, которые делают это:

```
g_pDMLoader->ReleaseObjectByUnknown(pDMSegment);
pDMSegment->Release();
```

Кроме того, если вы загружали коллекцию инструментов, пришло время выгрузить ее из объекта загрузчика, точно также, как вы это делали, когда освобождали музыкальный сегмент. Чтобы сделать очистку кэша более простой, предусмотрен единственный вызов, который можно использовать для принудительной очистки всего кэша. Вам не нужно выполнять этот вызов перед освобождением загрузчика, поскольку в этом случае все выполняется автоматически.

Вот как выглядит очистка кэша:

```
g_pDMLoader->ClearCache(GUID_DirectMusicAllTypes);
```

#### ПРИМЕЧАНИЕ

Вы должны выгружать данные кэша только когда уверены, что новый сегмент, который вы будете загружать не нуждается в кэшированной информации. Если вы загружаете отдельные песни, то выгрузка не представляет опасности.

## Изменение музыки

Вы можете выполнить с музыкой ряд действий, включая изменение уровня громкости, смену темпа и применение спец эффектов путем использования объекта звукового буфера DirectSound. Давайте взглянем на каждый из методов.

### Установка громкости

Вы можете менять два параметра громкости — общую громкость объекта исполнителя (и музыкальной системы в целом) и индивидуальную громкость воспроизведения отдельного сегмента. Как показано на рис. 4.16, каждый сегмент подвергается изменению громкости, когда передается объекту исполнителя. Объект исполнителя влияет на общую громкость.



*Рис. 4.16. Каждый сегмент может менять собственную громкость, а общая громкость влияет на все сегменты*

Громкость исполнителя (основная громкость) представляет собой глобальный параметр и для ее установки используется вызов **IDirectMusicPerformance8::SetGlobalParam**:

---

```
HRESULT IDirectMusicPerformance8::SetGlobalParam(
    REFGUID rguidType, // Устанавливаемый глобальный параметр
    void *pParam,      // Новое значение параметра
    DWORD  dwSize);    // Размер данных параметра
```

Параметр **rguidType** — это GUID глобального параметра, который нужно установить, в данном случае **GUID\_PerfMasterVolume**. Вы можете менять много глобальных параметров, так что за дополнительной информацией обращайтесь к документации DX SDK.

Параметр **pParam** — это уровень громкости, который вы хотите установить. Значение **dwSize** — это размер длинного целого, то есть размер переменной, которую вы используете для хранения уровня громкости. DirectMusic использует два макроса с именами **DMUS\_VOLUME\_MIN** (–200 децибел) и **DMUS\_VOLUME\_MAX** (+20 децибел), представляющих минимальный и максимальный уровень громкости соответственно. Используя значения, расположенные между значениями этих двух макросов, вы можете задать степень затухания в децибелах.

Вы можете упростить задание уровня громкости, создав формулу, которая позволит использовать для задания уровня громкости проценты, а не абсолютные значения. Проценты меняются в диапазоне от 0 до 100, где 0 представляет отсутствие звука, а 100 — максимально усиленный звук. Верно — максимальный уровень громкости усиливает звук (и слегка искажает его!), так что убедитесь, что выставили требуемый уровень.

Вот небольшая функция, которую вы можете использовать для указания основного уровня громкости, задавая значения в процентах от 0 до 100:

```
BOOL SetMasterVolume(long Level)
{
    long Volume, Range;

    // Вычисляем диапазон уровней громкости
    // и формируем новое значение
    Range = labs(DMUS_VOLUME_MAX - DMUS_VOLUME_MIN); // 220
    Volume = DMUS_VOLUME_MIN + Range / 100 * Level;

    // Устанавливаем новый уровень громкости
    if (FAILED(g_pDMPPerformance->SetParam(GUID_PerfMasterVolume,
                                             &Volume, sizeof(long))))
        return FALSE;

    return TRUE;
}
```

Установка громкости музыкального сегмента выполняется путем перехвата интерфейса аудио-пути и его последующего использования для установки новой громкости. Поскольку вы уже создали аудио-путь по умолчанию, получение указателя заключается в простом обращении к следующей функции:

```
HRESULT IDirectMusicPerformance8::GetDefaultAudioPath(
    IDirectMusicAudioPath8 **ppAudioPath);
```

Функция получает единственный параметр — указатель на используемый вами объект **IDirectMusicAudioPath8**. Когда указатель на аудио-путь получен, можно воспользоваться функцией **IDirectMusicAudioPath8::SetVolume**:

```
HRESULT IDirectMusicAudioPath8::SetVolume(  
    long lVolume,          // Устанавливаемый уровень громкости  
    DWORD dwDuration);    // Время выполнения изменения (миллисекунды)
```

Уровень громкости варьируется от -600 (тишина) до 0 (полная громкость). Усиления здесь нет. Чтобы излишне не нагружать процессор, время выполнения изменения должно быть равно 0. Присваивание **dwDuration** значения 0 также сообщает музыкальной системе о том, что громкость надо изменить немедленно.

Почему бы не задавать значение в процентах, вместо того, чтобы задавать громкость числами из диапазона от -600 до 0? Вы можете создать простую функцию, такую как я сейчас покажу вам, для вычисления уровня громкости. Используя этот уровень громкости вы можете получить объект аудио-пути и изменить громкость, как показано ниже:

```
BOOL SetSegmentVolume(IDirectMusicSegment8 *pDSegment,  
                      long Level)  
{  
    long Volume;  
    IDirectMusicAudioPath8 *pDMAudioPath;  
  
    // Получаем объект аудио-пути для работы с ним  
    if(FAILED(g_pDMPPerformance->GetDefaultAudioPath(  
                                                &pDMAudioPath)))  
        return FALSE;  
  
    // Вычисляем значение громкости для использования  
    // в объекте аудио-пути  
    Volume = -96 * (100 - Level);  
    HRESULT Result = pDMAudioPath->SetVolume(Volume, 0);  
  
    // Освобождаем аудио-путь - мы закончили работать с ним  
    pDMAudioPath->Release();  
  
    // Возвращаем флаг успеха (TRUE) или неудачи (FALSE)  
    // в зависимости от кода HRESULT возвращенного SetVolume  
    if(FAILED(Result))  
        return FALSE;  
  
    return TRUE;  
}
```

## **Смена темпа**

Вообразите себе возможность небольшого изменения темпа музыки, оживляющего игровой процесс и сообщающего игроку о том, что сейчас происходит. Например, когда игрок приближается к разъяренному монстру, темп ускоряется, и мы знаем, что нас ждут неприятности.

Темп измеряется в *ударах в минуту* (*beats per minute, BPM*), и обычное значение равно 120. DirectMusic позволяет менять темп различными способами. Простейший путь — настроить *основной темп исполнителя* (*performance master tempo*) путем задания коэффициента масштабирования. Например, коэффициент масштабирования 0.5 вдвое замедлит темп, а коэффициент 2.0 удвоит его.

Это выполняется путем установки глобального параметра, которую вы уже делали. На этот раз, однако, вы меняете параметр **GUID\_PerfMasterTempo**, указывая коэффициент масштабирования, являющийся числом типа **float**. Вот небольшая вспомогательная функция для этого. Я сделал ее так, что вы задаете значение в процентах, а не коэффициент масштабирования, что несколько упрощает задание темпа:

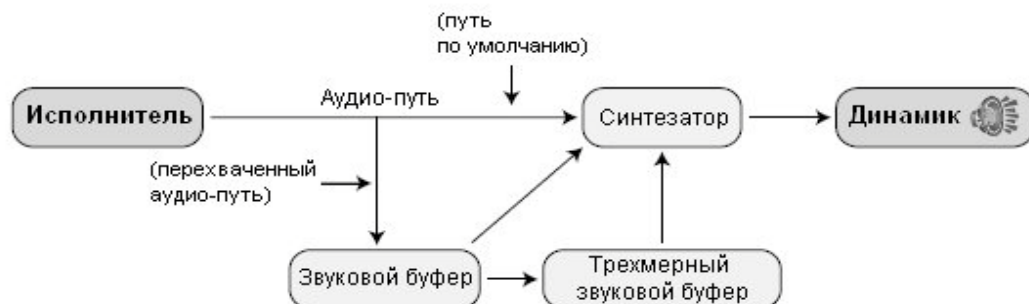
```
BOOL SetTempo(long Percent)
{
    float Tempo;

    Tempo = (float)Percent / 100.0f;
    if(FAILED(g_pDMPPerformance->SetGlobalParam(
        GUID_PerfMasterTempo,
        (void*)&Tempo, sizeof(float)))
        return FALSE;
    return TRUE;
}
```

Единственная загвоздка здесь в том, что проявление эффекта смены темпа может занять пару секунд из-за синхронизации такта. Также помните, что **SetTempo** влияет на общий темп, и меняться будет темп всех воспроизводимых сегментов. Завершив работу с песней вы должны вернуть нормальное значение темпа (1.0 или 100%).

## Захват аудио-канала

Последним в длинном списке музыкальных возможностей идет захват объекта звукового буфера DirectSound, используемого для синтеза инструментов и музыки. Вы можете сделать это получив объект аудио-пути и потом воспользовавшись им для захвата интерфейса звукового буфера.



**Рис. 4.17.** Перехват аудио-потока позволяет использовать буферы DirectSound (обычные и трехмерные) для создания некоторых ошеломительных эффектов



На рис. 4.17 показан устанавливаемый по умолчанию поток данных по аудио-пути от объекта исполнителя к синтезатору. Вы перехватываете этот поток и меняете его как хотите.

Все это назначение функции **IdirectMusicAudioPath8::GetObjectInPath**:

```
HRESULT IDirectMusicAudioPath8::GetObjectInPath(
    DWORD    dwPChannel,    // DMUS_PCHANNEL_ALL (каналы для поиска)
    DWORD    dwStage,       // DMUS_PATH_BUFFER (этап пути)
    DWORD    dwBuffer,       // 0 (индекс в цепочке буферов)
    REFGUID   guidObject,    // GUID_NULL (класс объекта)
    DWORD    dwIndex,        // 0 (индекс объекта в буфере)
    REFGUID   iidInterface,  // GUID желаемого объекта
    void      **ppObject);   // Указатель на создаваемый объект
```

Чтобы захватить объект звукового буфера вам достаточно указать его GUID и предоставить инициализируемый указатель. Показанная ниже функция получает заданный по умолчанию путь из объекта исполнителя и предоставляет вам объект **IDirectSoundBuffer8**, с которым вы можете экспериментировать:

```
IDirectSoundBuffer8 *GetSoundBuffer()
{
    IDirectMusicAudioPath8 *pDMAudioPath;
    IDirectSoundBuffer      *pDSB;
    IDirectSoundBuffer8     *pDSBuffer;

    // Получаем аудио-путь по умолчанию
    if (FAILED(g_pDMPPerformance->GetDefaultAudioPath(
                                                &pDMAudioPath)))
        return NULL;

    // Создаем объект IDirectSoundBuffer
    // и освобождаем объект аудио-пути
    if (FAILED(pDMAudioPath->GetObjectInPath(DMUS_PCHANNEL_ALL,
                                            DMUS_PATH_BUFFER, 0,
                                            GUID_NULL, 0,
                                            IID_IDirectSoundBuffer,
                                            (LPVOID*)&pDSB))) {
        pDMAudioPath->Release();
        return FALSE;
    }
    pDMAudioPath->Release();

    // Запрашиваем новый объект звукового буфера
    // и возвращаем его
    if (FAILED(pDSB->QueryInterface(IID_IDirectSoundBuffer8,
                                   (void**)&pDSBuffer))) {
        pDSB->Release();
        return FALSE;
    }
    pDSB->Release();
    return pDSBuffer;
}
```

Вы можете изменить показанную выше функцию для запроса интерфейса трехмерного звукового буфера просто изменив идентификатор

---

интерфейса на `IID_IDirectSound3DBuffer8`. Убедитесь, что завершив работу вы освобождаете этот новый объект.

## Присоединяемся к MP3-революции

Если последние несколько лет вы жили в пещере, то могли пропустить появление разработанного в институте Фраунгофера формата музыкальных файлов MP3, который вызвал громадные потрясения в мире музыки. Куда ни повернешься можно увидеть портативные плееры, домашние кинотеатры, автомобильные магнитолы — и все они поддерживают воспроизведение музыки в формате MP3.

Так что же такое MP3? Это формат сжатия аудиоданных, позволяющий сделать их намного компактнее. Я мог бы долго продолжать разговор о том, что делает формат MP3 таким особенным, но в действительности он настолько вошел в нашу жизнь, что вы должны были бы жить в пещере, если не знаете что он делает для вас — воспроизводит музыку.

Возможность использовать MP3 в ваших проектах дает Microsoft DirectShow. Всего несколько коротких строк кода позволят вам воспроизводить в вашем проекте любой MP3-файл (или любой другой мультимедийный формат, DirectShow поддерживает WMA, AVI, MPG и это только несколько названий). Единственное требование, чтобы в вашей системе был установлен необходимый кодек (большинство популярных кодеков распространяются вместе с Windows 98 и XP).

---

**ПРИМЕЧАНИЕ**

*Кодек (это означает кодирование/декодирование) — это процесс или программа, используемая для зашифровывания и/или расшифровывания определенного типа формата данных, например, MP3. Обычно вы получаете кодек от компании, разработавшей формат данных. В частности, кодек MP3 предоставляется институтом Фраунгофера. К счастью Windows поставляется вместе с набором кодеков для наиболее популярных форматов — MP3, AVI, MPG и т.д., так что вам не надо беспокоиться об их поиске в Интернете.*

---

---

**ПРИМЕЧАНИЕ**

*Чтобы использовать DirectShow в собственных проектах необходимо добавить к проекту заголовочный файл dshow.h и библиотеку strmiids.lib.*

---

Давайте посмотрим, как настроить DirectShow для воспроизведения звуков и музыки в вашем проекте.

## Использование DirectShow

Как и везде в DirectX, для доступа к компонентам DirectShow используются СОМ-объекты и интерфейсы. Интерфейсы с которыми мы будем иметь дело перечислены в таблице 4.5.

Таблица 4.5. COM-интерфейсы DirectShow

Интерфейс	Описание
<b>IGraphBuilder</b>	Помогает построить граф фильтров. Граф фильтров — это набор объектов и интерфейсов, используемых для обработки медиафайлов.
<b>IMediaControl</b>	Управляет потоком данных через граф фильтров. Этот интерфейс вы используете для контроля воспроизведения ваших звуков и музыки.
<b>IMediaEvent</b>	Получает уведомления о событиях от графа фильтров. Это полезно, если вы хотите знать, что происходит в вашем графе фильтров — воспроизводится ли еще музыка, остановлена ли она и т.д.

Хотя в таблице 4.5 перечислено лишь несколько интерфейсов, их достаточно для воспроизведения MP3-файлов в ваших проектах. Первый из этих трех объектов, **IGraphBuilder**, большая шишка среди объектов DirectShow — он создает набор фильтров, обрабатывающих медиаданные и делает другие полезные вещи. В нашем случае это загрузка MP3-файла, инициализация декодера и передача звука на ваши динамики.

Звучит сложно, но делается просто. Первый этап — создание объекта построителя графа с помощью функции **CoCreateInstance**:

```
STDAPI CoCreateInstance(
    REFCLSID rclsid,          // ID класса (CLSID) желаемого объекта
    LPUNKNOWN pUnkOuter,     // NULL
    DWORD dwClsContext,      // Контекст класса (CLSCTX_INPROC_SERVER)
    REFIID riid,             // Ссылка на идентификатор
    LPVOID *ppv);            // Указатель на создаваемый объект
```

**CoCreateInstance** исключительно полезная функция; она позволяет вам создать любой COM-объект, задав идентификатор класса и ссылку на идентификатор объекта. В данном случае идентификатор класса будет **CLSID\_FilterGraph**, а идентификатор объекта — **IID\_IGraphBuilder**.

```
IGraphBuilder *pGraphBuilder = NULL;

if (FAILED(CoCreateInstance(CLSID_FilterGraph, NULL,
                           CLSCTX_INPROC_SERVER,
                           IID_IGraphBuilder,
                           (void**) &pGraphBuilder))) {
    // Произошла ошибка
}
```

После того, как вы использовали **CoCreateInstance** для создания вашего объекта построителя графа, вы можете запросить два медиа-объекта, перечисленных в таблице 4.5:

**ВНИМАНИЕ!**

Обязательно убедитесь, что инициализировали COM-систему перед вызовом `CoCreateInstance`. Для приложений, которые не используют несколько потоков (однопоточных), добавьте следующую строку до того, как делать какие-либо вызовы DirectX:

```
CoInitialize(NULL);
```

С другой стороны, если вы используете многопоточность, воспользуйтесь следующим вызовом функции:

```
CoInitializeEx(NULL, COINIT_MULTITHREADED);
```

Убедитесь, что при завершении работы ваше приложение вызывает функцию `CoUninitialize`:

```
CoUninitialize();
```

```
IMediaControl *pMediaControl = NULL;
IMediaEvent *pMediaEvent = NULL;

pGraphBuilder->QueryInterface(IID_IMediaControl,
                              (void**)&pMediaControl);
pGraphBuilder->QueryInterface(IID_IMediaEvent,
                              (void**)&pMediaEvent);
```

Итак, теперь когда у вас есть три интерфейса для работы, что мы будем делать с ними? Первая вещь — загрузка медиа-файла с которым вы хотите работать.

## Загрузка медиа-файла

Собственно говоря, вам не надо в действительности загружать медиа-файл, вы создаете ссылку из фильтра `DirectShow` на файл данных. Поток данных поступает из файла по мере расшифровки, что сокращает объем памяти, необходимый для воспроизведения содержимого. Процесс создания такой ссылки из медиа-файла к фильтру называется *визуализацией* (*rendering*).

Чтобы визуализировать файл вам надо всего лишь вызвать функцию **`IGraphBuilder::RenderFile`**:

```
HRESULT IGraphBuilder::RenderFile(
    LPCWSTR lpwstrFile,          // Имя обрабатываемого медиа-файла
    LPCWSTR lpwstrPlayList);    // Не используется - укажите NULL
```

По сути **`RenderFile`** требует от вас один параметр — имя загружаемого медиа-файла. (Второй параметр не используется, так что укажите в нем **`NULL`**.) Обратите внимание, что строка с именем файла имеет тип **`LPCWSTR`**, а это указатель на строку широких символов.

**ПРИМЕЧАНИЕ**

О работе со строками широких символов мы уже говорили в разделе «Создание объекта загрузчика» этой главы.

Вот, например, как можно визуализировать файл с именем `song.mp3`:

```
pGraphBuilder->RenderFile(L"song.mp3", NULL);
```

## Управление воспроизведением вашей музыки и звуков

После того, как медиа-файл визуализирован, вы можете перейти к управлению воспроизведением звука и определением состояния воспроизведения с использованием двух ранее созданных интерфейсов **IMediaControl** и **IMediaEvent**.

Первый объект, **IMediaControl**, вы используете для управлением воспроизведением медиа-файла. Здесь стоит уделить внимание трем функциям: **IMediaControl::Run**, **IMediaControl::Pause** и **IMediaControl::Stop**. У каждой из этих функций есть свое назначение и они рассматриваются в последующих подразделах.

### Воспроизведение звука

Чтобы начать воспроизведение вашего медиа-файла, вызовите функцию **IMediaControl::Run**:

```
HRESULT IMediaControl::Run();
```

У функции нет параметров, просто вызовите ее, чтобы началось воспроизведение вашей песни:

```
pMediaControl->Run();
```

### Приостановка воспроизведения

Получив воспроизводящийся поток данных вы можете позволить ему воспроизводиться дальше, сделать паузу или вообще прекратить воспроизведение. Чтобы приостановить воспроизведение звука используйте функцию **IMediaControl::Pause**:

```
HRESULT IMediaControl::Pause();
```

Еще одна не напрягающая мозги функция — для паузы в воспроизведении песни вызовите **Pause**:

```
pMediaControl->Pause();
```

Чтобы закончить паузу в воспроизведении просто вызовите снова **IMediaControl::Run**.

### Завершение воспроизведения

Имея воспроизводящийся медиа-файл вы можете не только приостановить его, но и полностью завершить воспроизведение, используя функцию **IMediaControl::Stop**:

```
HRESULT IMediaControl::Stop();
```

И снова здесь нет никаких параметров, так что использовать ее просто:

```
pMediaControl->Stop();
```

## Обнаружение событий

Теперь вы увидели, как управлять воспроизведением медиа-файла. Но что если вам необходимо узнать, что происходит с медиа-файлом во время воспроизведения? Как он может сообщить вам, что воспроизведение остановлено из-за достижения конца песни? С помощью интерфейса **IMediaEvent** — вот как!

**IMediaEvent** предоставляет три функции, которые интересны для нас:

- **GetEvent**
- **FreeEventParams**
- **WaitForCompletion**

Чаще всего вы будете иметь дело с первой функцией, **GetEvent**, которая получает события, сигнализирующие о состоянии воспроизведения музыки.

```
HRESULT IMediaEvent::GetEvent(
    long *lEventCode, // Код события
    long *lParam1,    // Параметр события 1
    long *lParam2,    // Параметр события 2
    long msTimeout);  // Время ожидания события
```

Для вызова **GetEvent** вам надо предоставить четыре параметра — указатель на переменную, которая будет содержать код события, два дополнительных указателя на переменные для хранения различных связанных с событием параметров и, наконец, период времени (в миллисекундах) в течение которого будет ожидаться возникновение события.

Одно из событий, которое вы наверняка будете отслеживать, — завершение воспроизведения. О нем сигнализирует значение **EC\_COMPLETE**. Так что, если после вызова **GetEvent** в **lEventCode** находится значение **EC\_COMPLETE**, вы знаете, что воспроизведение завершилось.

Вот пример использования **GetEvent**:

```
long Event, Param1, Param2;

// Ждем возникновения события 1 мс.
pMediaEvent->GetEvent(&Event, &Param1, &Param2, 1);
if(Event == EC_COMPLETE) {
    // Воспроизведение завершено
}
```

После того, как вы получили и обработали событие из **GetEvent**, необходимо освободить ресурсы, выделенные DirectShow, вызвав **IMediaEvent::FreeEventParams**:

```
HRESULT IMediaEvent::FreeEventParams(  
    long lEventCode, // Событие из GetEvent  
    long lParam1,    // Параметр 1 из GetEvent  
    long lParam2);   // Параметр 2 из GetEvent
```

Вы вызываете **FreeEventParams** используя те же переменные, которые получили при вызове **GetEvent**:

```
// Получение какого-либо события  
pMediaEvent->GetEvent(&Event, &Param1, &Param2, 1);  
  
// Обработка события  
  
// Освобождение ресурсов события  
pMediaEvent->FreeEventParams(Event, Param1, Param2);
```

Постоянно вызывая **GetEvent** и **FreeEventParams** в каждом кадре вы можете определить момент завершения воспроизведения вашей песни и предпринять необходимые действия. Но что, если вы просто хотите воспроизвести медиа-файл полностью, и не желаете постоянно следить за его завершением?

Вы можете использовать третью из упомянутых мной функций, которая контролирует воспроизведение медиа-файла за вас и возвращает управление, когда оно завершено. Это функция **IMediaEvent::WaitForCompletion**:

```
HRESULT IMediaEvent::WaitForCompletion(  
    long msTimeout, // Установите INFINITE  
    long *pEvCode); // Код события
```

Поскольку единственное событие, которого вы ждете, — это завершение воспроизведения, можно использовать следующий небольшой фрагмент кода для запуска воспроизведения медиа-файла и ожидания его завершения:

```
// Воспроизведение песни  
pMediaControl->Run();  
  
// Ждем завершения  
pMediaEvent->WaitForCompletion(INFINITE, &Event);  
  
// Выполняем действия, необходимые после  
// завершения воспроизведения
```

## Освобождение ресурсов DirectShow

После того, как вы полностью закончили работу с вашими медиа-файлами, пришло время все выключить. После остановки воспроизведения медиа-файла (через **IMediaControl::Stop**) вы можете освободить объекты DirectShow используя их функции **Release**:

```
pMediaEvent->Release(); pMediaEvent = NULL;  
pMediaControl->Release(); pMediaControl = NULL;  
pGraphBuilder->Release(); pGraphBuilder = NULL;
```

**ПРИМЕЧАНИЕ**

Позднее в этой книге, а именно в главе 6, вы увидите, что для безопасного освобождения COM-объектов и присваивания указателям значения `NULL`, я использую собственный макрос `ReleaseCOM`. Относящиеся к DirectX вспомогательные библиотеки Microsoft делают тоже самое с помощью макроса с именем `SAFE_RELEASE` (определенного в файле `dxutil.h`, поставляемом вместе с DirectX SDK). Вы больше узнаете о макросе `ReleaseCOM`, когда встретитесь с его использованием в 6 главе.

Вот и все, что относится к воспроизведению MP3-музыки в ваших проектах!

## Заканчиваем с музыкой

DirectX Audio — трудный клиент. В нем есть два сложных для работы компонента (DirectSound и DirectMusic), получив которые можно попытаться выяснить запутанные детали каждого из них. Вы создаете звуковые буферы и загружаете их звуковыми данными, отбираете данные инструментов, загружаете и воспроизводите музыкальные сегменты.

К счастью, в вашем распоряжении есть эта глава! Благодаря содержащейся в ней информации вы сможете использовать техники потокового воспроизведения, собственные DLS-инструменты и спецэффекты при воспроизведении, такие как изменение громкости, позиционирование и настройка частоты. Нельзя не упомянуть, что вы получили возможность воспроизводить MP3-музыку, используя исключительно простой DirectShow.

Эта глава охватывает много информации, которую вы найдете исключительно полезной для разработки своих собственных программ. Кроме того, глава 6 расскажет вам как использовать эти техники для создания библиотеки удобных звуковых и музыкальных функций.

### Программы на CD-ROM

Программы, демонстрирующие обсуждавшийся в этой главе код, находятся на прилагаемом к книге CD-ROM. В папке `\BookCode\Chap04\` вы найдете следующие программы:

**Shell** — каркас приложения, выполняющий инициализацию DirectSound и DirectMusic. Местоположение: `\BookCode\Chap04\Shell\`.

**LockLoad** — программа создает звуковой буфер, блокирует его и заполняет случайными данными. Местоположение: `\BookCode\Chap04\LockLoad\`.

**WavPlay** — программа загружает небольшой волновой файл (.WAV) и воспроизводит его. Местоположение: `\BookCode\Chap04\WavPlay\`.



### **Программы на CD-ROM (продолжение)**

**Stream** — программа загружает один волновой файл (.WAV) и использует потоковую технику для его воспроизведения.  
Местоположение: \BookCode\Chap04\Stream\.

**MidiPlay** — программа загружает MIDI-файл и воспроизводит его.  
Местоположение: \BookCode\Chap04\MidiPlay\.

**MP3Play** — программа загружает и воспроизводит файл MP3.  
Местоположение: \BookCode\Chap04\MP3Play\.

# Глава 5

## Работа с сетью с DirectPlay

Я помню ночи, которые провел за компьютером уничтожая орды монстров, ища сокровища и просто хорошо проводя время со своими Интернет-приятелями. Ха, да что я говорю; я так поступаю и сейчас.

Интернет-игры (и сетевые игры) изменили мир, сделав возможность многопользовательской игры стандартом. Игроки больше не ограничены выбиванием пыли из компьютера; теперь они оттачивают свои умения друг на друге. Чтобы у вашей игры был шанс выжить на современном требовательном рынке, необходимо предоставить игрокам такую возможность. Вы можете добавить поддержку сети к растущему списку возможностей вашей игры с помощью Microsoft DirectPlay.

В этой главе вы узнаете о следующих вещах:

- Основы работы с сетями.
- Работа с интерфейсами DirectPlay.
- Обработка сетевых сообщений.
- Работа с серверами и клиентами.

### Знакомство с сетями

*Сеть (network)* — это несколько компьютеров, соединенных друг с другом для обмена данными и совместной работы. Помимо нескольких компьютеров для сети требуется сетевое программное обеспечение (или *сетевая операционная система*), сетевые адаптеры и кабели. Сетевые адаптеры бывают разных размеров и форм, но обычно встречаются адаптеры в виде модема. Верно, ваш модем — это сетевой адаптер, дающий возможность соединиться с миллионами других компьютеров по всему миру в величайшей сети, Интернете.

В данной главе я сосредоточусь на работе с сетью с точки зрения игрового процесса и программирования игр. На первый взгляд работа с сетью может выглядеть пугающе. Множество компьютеров, передающих бесконечные потоки данных, как и что можно сделать, чтобы разобраться во всем этом? Как и со многими другими вещами, следует начать с начала и

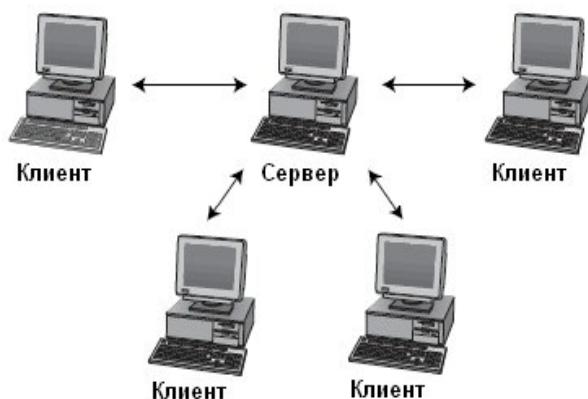
изучить концепции, лежащие в основе сетей; затем вы разовьете ваши знания, основываясь на реальных (и простых) примерах.

## Сетевые модели

Существует две основных сетевых модели: *клиент-сервер* (*client-server*) и *одноранговая* (*peer-to-peer*). Каждая модель в той или иной форме объединяет компьютеры для совместного пользования информацией. Какую модель использовать зависит от вас, но принимать решения следует основываясь на ваших потребностях, поскольку у каждой модели есть свои сильные и слабые стороны.

Сервер используется при создании централизованной сети. Остальные компьютеры являются клиентами и подключаются к серверу, обмениваясь информацией только с ним.

У клиента нет прямых связей с другими клиентами; он знает только о сервере. Сервер знает обо всех клиентах и маршрутизирует данные между ними как считает нужным. На рис. 5.1. показаны взаимосвязи между компьютерами в модели клиент-сервер.



**Рис. 5.1.** Клиенты могут подключаться к серверу, но ничего не знают друг о друге

---

### ПРИМЕЧАНИЕ

Пара из сервера и клиента обычно описывается как общая модель клиент/сервер. Однако, при использовании DirectPlay необходимо разделение на модель сервера и модель клиента, поскольку они представляются двумя отдельными компонентами.

---

Другая модель сети — одноранговая, в которой компьютеры соединяются друг с другом напрямую. Новая связь устанавливается для каждого нового компьютера, подключающегося к сессии, поэтому каждый компьютер знает обо всех других. Как показано на рис. 5.2, для сети из четырех компьютеров необходимо 12 соединений (помните, что у каждого компьютера есть подключения к каждому из трех других компьютеров).

---

### ПРИМЕЧАНИЕ

Период времени, в течение которого вы подключены к сети, называется *сессией* (*session*). С сессией могут быть связаны различные свойства, такие как пароль, максимальное число подключений и т.д. В главе 15, «Реализация многопользовательского режима», вы узнаете как эта информация относится к играм.

---



*Рис. 5.2. В одноранговой сети каждый компьютер соединен со всеми другими компьютерами сети*

#### ПРИМЕЧАНИЕ

Некоторые игры используют одноранговую модель, поскольку позволяют одновременную игру только четырех или восьми игроков. Примером может служить Diablo (от Blizzard Entertainment), поддерживающая до четырех игроков. С другой стороны, такие игры как Ultima Online (от Origin Systems) или EverQuest (от Sony) поддерживают тысячи игроков, используя модель клиент/сервер.

Какой тип сетевой модели лучше подходит вам, зависит от используемого приложения. Модель клиент/сервер лучше подходит для сетей в которых больше четырех пользователей, а одноранговая модель лучше приспособлена для прямого соединения или небольших домашних сетей.

Главной проблемой при большом количестве компьютеров в сети становится попытка найти других игроков. Хотя вы можете как обычно играть с несколькими друзьями (и, конечно, знать, как найти их), что делать, если в игру хочется пригласить кого-нибудь нового? В большинстве случаев для этих целей сетевые игры используют лобби-серверы.

## Лобби

Вы можете думать о лобби-сервере (*lobby server*), как о зале для встреч игроков. Лобби-сервер позволяет игрокам регистрироваться, общаться и вместе играть в их любимые игры. Как только лобби-сервер соединяет игроков, он выходит из цикла обмена данными (для экономии полосы пропускания сети).

---

<b>ПРИМЕЧАНИЕ</b>	<i>Полоса пропускания сети (network bandwidth) определяет объем данных, который может пройти через сетевое соединение без затруднений. Широкополосные соединения обрабатывают большие объемы данных и быстрее, чем узкополосные.</i>
-------------------	--

---

Замечательный лобби-сервер расположен на сайте Microsoft Gaming Zone (по адресу <http://zone.msn.com>). Он позволяет тысячам игроков общаться друг с другом, а затем соединяет их напрямую, чтобы они играли в свои любимые сетевые игры. Gaming Zone поддерживает множество игр, и если ваша игра окажется популярной, там может появиться и ее поддержка!

Хотя лобби-серверы и полезны для многопользовательских игр, их обсуждение выходит за рамки этой книги. Документация DirectX SDK содержит много информации об использовании лобби-серверов, а в SDK включены несколько примеров приложений с поддержкой лобби-серверов, которые вы можете посмотреть. За дополнительной информацией об этих приложениях обращайтесь к DX SDK (вы найдете его на прилагаемом к книге CD-ROM).

## Задержка и запаздывание

Ширина полосы пропускания приносит два новых термина: задержка и запаздывание. *Задержка (latency)* — это количество времени, необходимое для выполнения операции (чем меньше, тем лучше). Термин *запаздывание (lag)* используется для описания задержек в сетевых коммуникациях — времени, проходящего с того момента, как данные отправлены, до того момента, когда они будут получены.

Малое запаздывание означает быструю передачу данных. Большое запаздывание (очень неприятная вещь) означает, что сетевые данные задерживаются или вообще не будут доставлены. Запаздывание — это большая проблема, особенно когда работаешь с Интернетом, так что вы будете иметь дело с ним.

## Коммуникационные протоколы

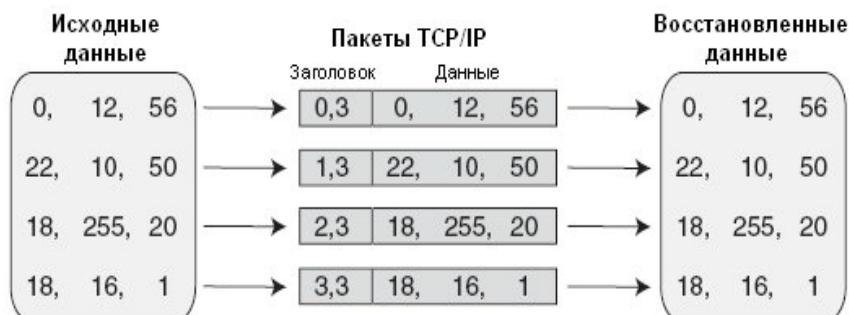
Компьютеры в сети могут взаимодействовать между собой различными способами, но чтобы понимать друг друга две системы должны использовать один и тот же протокол. На данный момент наиболее популярным протоколом является TCP/IP (Transfer Control Protocol/Internet Protocol), используемый в Интернете, и только с ним мы будем иметь дело в этой книге.

---

<b>ПРИМЕЧАНИЕ</b>	Коммуникационные протоколы иногда называют <i>поставщиками услуг (service provider)</i> . Думайте о поставщике услуг как о вашем интерфейсе к сети, будь это протокол, такой как IPX или TCP/IP, или устройство, такое как модем или последовательный кабель.
-------------------	---

---

Протокол TCP/IP — это метод упаковки данных и их отправки через сеть. Он делает это разделяя данные на небольшие пакеты и добавляя к ним адреса отправителя и получателя, а также номер пакета, используемый для воссоздания первоначальных данных (как показано на рис. 5.3). Эти пакеты отправляются в великое неизведанное с надеждой, что они доберутся до места назначения.



**Рис. 5.3.** TCP/IP разделяет данные на пакеты и добавляет к ним свою собственную информацию заголовка. Заголовок содержит номер пакета, адрес отправителя и адрес назначенной цели

TCP/IP позволяет повторно отправлять пакеты по сети, на случай если какая-то информация потеряется во время передачи (такое случается достаточно часто). Когда возникает запаздывание, пакеты могут даже приниматься в неправильном порядке, и старые пакеты будут получены после новых. Не стоит беспокоиться; TCP/IP заботится о повторной отправке потерянных пакетов и переупорядочивании пакетов, полученных в неверной последовательности.

## Адресация

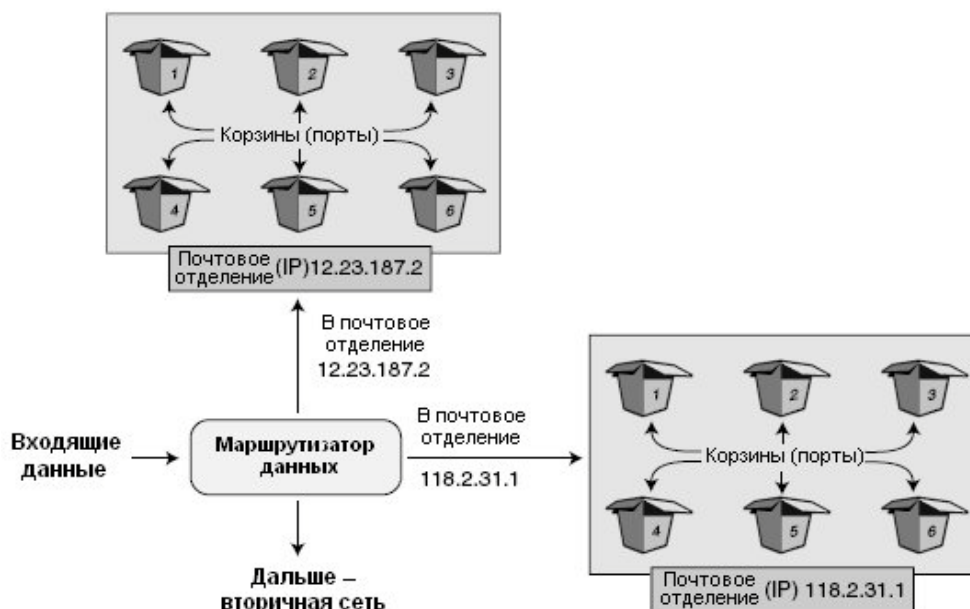
В Интернете столько компьютеров, как же данные знают, куда направляться? Точно также, как вы пишете на конверте адрес, перед тем как бросить его в почтовый ящик, протокол TCP/IP назначает каждой системе сетевой адрес (*IP-адрес*), состоящий из разделенных точками четырех чисел (которые могут быть в диапазоне от 0 до 255). IP-адрес может выглядеть так:

64.120.53.2

Хотя для человека адрес выглядит тарабарщиной, сеть может успешно направлять данные, основываясь на этих значениях. Если вы любите математику, можете посчитать, что комбинация четырех чисел дает 4 294 967 296 возможных адресов. Чтобы увеличить количество адресов, в сетях используются дополнительные значения, называемые *порты* и определяющие куда должны быть доставлены данные.

Думайте об IP-адресе, как о почтовом отделении. Такое почтовое отделение (IP-адрес) представляет отдельный компьютер, подключенный к сети, и ему назначен единственный IP-адрес. Внутри почтового отделения есть много ящиков (или портов) по которым сортируется почта. Каждый ящик (порт) относится к отдельному офису (отдельному приложению).

Некоторые приложения используют несколько портов. Данные будут получены только той системой, которая имеет тот IP-адрес, которому они направлены, и знает какой порт надо прослушивать. Устройство, называемое *маршрутизатор данных (data router)* направляет входящие сетевые данные тем системам, которые оно знает, либо переправляет их другому сетевому соединению (называемому *вторичная сеть, pass-along network*). На рис. 5.4 показаны пути прохождения данных через маршрутизатор.



**Рис. 5.4.** У почтового отделения (IP-адреса) есть множество ящиков (портов) для сортировки входящих данных. Несколько почтовых отделений могут быть соединены через маршрутизатор данных

## Введение в DirectPlay

DirectPlay — это решение Microsoft для работы с сетями. Хотя в нескольких последних версиях DirectPlay подвергался серьезным изменениям, кажется, что в версии 8 Microsoft удалось реализовать действительно простую для использования систему. Фактически, настолько простую, что вы начнете работать с сетью прежде, чем узнаете это.

---

**ПРИМЕЧАНИЕ** Чтобы использовать DirectPlay в ваших проектах, убедитесь, что включили заголовочные файлы Dplay8.h и DPAddr.h, а также указали в списке компоновки библиотеки DPlay.lib и DXGuid.lib.

---



---

**ПРИМЕЧАНИЕ** Я знаю, о чем вы думаете — что делает весь этот материал о DirectX версии 8 в книге, посвященной использованию DirectX версии 9?! Не беспокойтесь, мой друг, Microsoft решила оставить неизменной 8 версию интерфейсов в DirectX SDK версии 9 (девятой версии интерфейсов не существует), так что информация, которую вы найдете здесь, применима к DirectX как 8, так и 9 версий.

---

## Сетевые объекты

Используя DirectPlay вы получаете доступ к трем, упомянутым ранее сетевым моделям: клиентской, серверной и одноранговой. У каждой есть свой объект интерфейса (все они описаны в таблице 5.1), и все предоставляют похожие функции.

**Таблица 5.1.** COM-объекты DirectPlay

<b>Объект</b>	<b>Описание</b>
<b>IDirectPlay8Client</b>	Объект клиента сети. Устанавливает соединение с сервером.
<b>IDirectPlay8Server</b>	Объект сервера. Устанавливает соединение с клиентом.
<b>IDirectPlay8Peer</b>	Объект одноранговой сети. Устанавливает соединения с другими одноранговыми клиентами.
<b>IDirectPlay8Address</b>	Объект, который хранит (и создает) сетевой адрес.

**ПРИМЕЧАНИЕ** Крупномасштабные ролевые игры обычно для эффективной работы требуют использования модели клиент/сервер, поэтому я не буду в этой книге обсуждать одноранговые сети. За примерами их использования обратитесь к документации DirectX SDK и книгам, перечисленным в приложении А. Я рекомендую в качестве источника информации об использовании одноранговых сетей в играх книгу Тодда Барона «Программирование многопользовательских игр».

Чтобы подключиться к удаленной системе (или запустить ведущий узел) вы конструируете сетевой адрес, используя интерфейс **IDirectPlay8Address** (показанный в таблице 5.1). Его единственная задача — создать и хранить единственный сетевой адрес.

**ПРИМЕЧАНИЕ** Как я упоминал ранее, сессия — это период времени в течение которого работает ведущий узел, либо вы подключены к удаленной системе. Когда соединение разрывается, сессия завершается. У каждой сессии есть уникальные свойства, такие как имя, пароль (если необходим), максимально возможное число пользователей и т.д. Более подробно о сессиях мы поговорим в разделе «Конфигурирование данных сессии» далее в этой главе.

Затем вы создаете сетевой объект и назначаете ему адрес, после чего все готово для запуска ведущего узла или подключения к удаленной системе. Запустив ведущий узел игры вы просто ждете, когда другие системы (то есть люди, использующие эти компьютеры) подключатся к вам, после чего ваша система и удаленный компьютер начинают обмениваться между собой



относящимися к игре сетевыми сообщениями. В DirectPlay такие удаленные системы (а также ваш компьютер) называются *игроками (players)*.

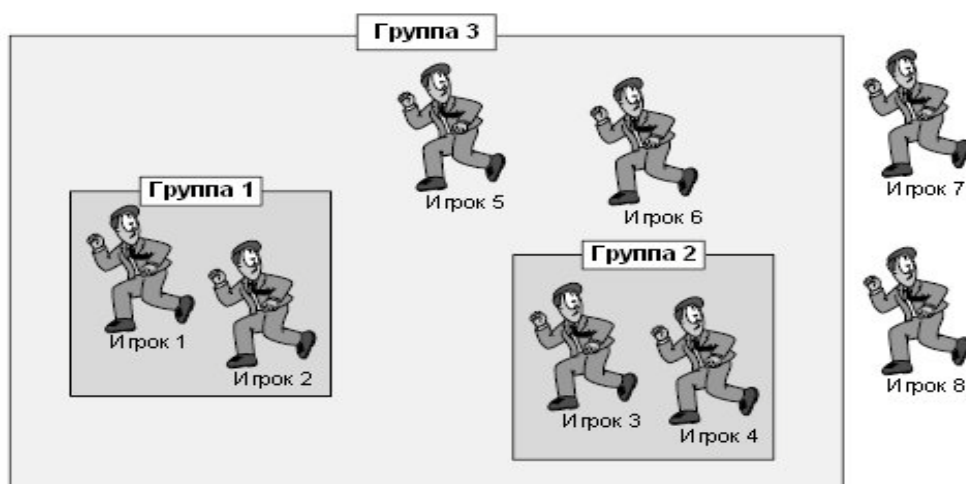
## Работа с игроками

В терминах DirectPlay *игрок (player)* — это отдельное подключение, соединяющее вас с другим компьютером в сети. На одном компьютере может быть несколько игроков, но обычно используется только один. Фактически, серверу для идентификации также назначается игрок.

Каждый игрок получает *идентификатор (Player ID)*, который система использует для направления сообщений отдельному игроку. Эти идентификаторы — единственный надежный метод определения игроков, так что ваша программа должна отслеживать их.

В больших играх могут быть тысячи подключенных игроков. Чтобы улучшить работу с игроками, некоторые (или все) игроки могут быть объединены в *группы (groups)*. Используя группы вы слегка упрощаете программирование, поскольку можете объединить в группу игроков, находящихся в одной локации игры (такой, как карта или уровень) и отправлять данные всей группе сразу, а не каждому игроку индивидуально. (Есть ряд других причин для использования групп, но эта — одна из наиболее важных.)

Взаимоотношения между игроками и группами показаны на рис. 5.5. Нет никаких ограничений на количество игроков в группе и на количество групп, которые вы можете создать. Как видно на рис. 5.5, группы (обозначенные прямоугольниками) могут входить в другие группы (группы 1 и 2 входят в группу 3). Обратите внимание, что игроки 7 и 8 изолированы от групп; для DirectPlay не имеет значения, входит игрок в группу или нет.



**Рис. 5.5.** Восемь игроков подключены к сеансу игры. Шесть игроков относятся к группе (или к двум группам), а два игрока находятся отдельно от всех групп

Независимо от того, используете вы группы или нет, системы игроков, как только установлено соединение, могут взаимодействовать друг с другом через сообщения.

## Сетевые сообщения

*Сообщение (message)* — это категоризированный пакет данных, завернутый в простую структуру. У каждого сообщения есть определенный смысл, ему назначен макрос (они показаны в таблице 5.2) и оно зависит от используемой сетевой модели. Например, объект клиента никогда не получит сообщение, предназначенное для однорангового объекта.

**Таблица 5.2.** Стандартные сообщения DirectPlay

<i>Макрос сообщения</i>	<i>Описание</i>
DPN_MSGID_ADD_PLAYER_TO_GROUP	Игрок добавлен в существующую группу.
DPN_MSGID_APPLICATION_DESC	Запрошены данные приложения.
DPN_MSGID_ASYNC_OP_COMPLETE	Асинхронная операция отправки данных завершена.
DPN_MSGID_CLIENT_INFO	Запрошены данные клиента.
DPN_MSGID_CONNECT_COMPLETE	Сетевое подключение завершено.
DPN_MSGID_CREATE_GROUP	Создана группа.
DPN_MSGID_CREATE_PLAYER	Создан игрок.
DPN_MSGID_DESTROY_GROUP	Группа уничтожена (удалена).
DPN_MSGID_DESTROY_PLAYER	Игрок уничтожен (удален).
DPN_MSGID_ENUM_HOSTS_QUERY	Сетевое приложение ищет другие для соединения.
DPN_MSGID_ENUM_HOSTS_RESPONSE	Сообщение позволяет вам отвечать на запрос хоста.
DPN_MSGID_GROUP_INFO	Запрошены данные группы.
DPN_MSGID_HOST_MIGRATE	Хост перемещает свои данные на другую систему из-за потери соединения.
DPN_MSGID_INDICATE_CONNECT	Удаленная система пытается подключиться.
DPN_MSGID_INDICATED_CONNECT_ABORTED	Подключенная удаленная система пытается разорвать соединение.
DPN_MSGID_PEER_INFO	Запрошены одноранговые данные.
DPN_MSGID_RECEIVE	Данные получены.
DPN_MSGID_REMOVE_PLAYER_FROM_GROUP	Игрок удален из группы.

**Таблица 5.2.** Стандартные сообщения DirectPlay (продолжение)

<i><b>Макрос сообщения</b></i>	<i><b>Описание</b></i>
<code>DPN_MSGID_RETURN_BUFFER</code>	DirectPlay завершил работу с предоставленным ему буфером.
<code>DPN_MSGID_SEND_COMPLETE</code>	Данные успешно отправлены.
<code>DPN_MSGID_SERVER_INFO</code>	Запрошены данные сервера.
<code>DPN_MSGID_TERMINATE_SESSION</code>	Сетевая сессия прервана.
<hr/>	
<b>ПРИМЕЧАНИЕ</b>	Хотя смысл некоторых сообщений сейчас может быть вам непонятен, описания дают отправную точку. Мы будем работать лишь с несколькими сообщениями, но это тот случай, когда чем меньше, тем лучше.
<hr/>	
<b>ПРИМЕЧАНИЕ</b>	Описание сетевых сообщений DirectPlay и связанной с ними информации вы найдете в документации к DirectX SDK. Я рекомендую, чтобы создавая свой шедевр, вы держали этот документ открытым под рукой.

Для получения сообщений вашему сетевому объекту должна быть назначена функция обратного вызова, которая вызывается каждый раз при поступлении сообщения. Чтобы обеспечить плавность потока данных эта функция выполняет разбор данных, основываясь на их типе, и выполняет возврат управления так быстро, как возможно.

Для отправки сообщений вы используете функцию отправки соответствующего сетевого объекта (их всего два). Эти функции просты в использовании и предоставляют вам множество параметров доставки, включая гарантированную доставку, безопасное шифрование, асинхронную и синхронную доставку.

### ***Асинхронная и синхронная работа***

Первый вариант доставки, предлагаемый вам DirectPlay — это *синхронная* или *асинхронная* отправка сообщений. Это означает, что либо система возвращает вам управление после того, как получила команду отправить данные (асинхронная работа), либо она ждет, пока данные не будут успешно отправлены (синхронная работа).

Что лучше? Скорее всего, вы предпочтете асинхронный метод, поскольку он не задерживает работу системы, как это делает синхронный. Например, если вы хотите отправить большой объем данных и одновременно пытаетесь играть в игру, то не захотите прерывать процесс, ожидая, пока сеть попытается передать информацию. Просто сообщите, что нужно отправить, и позвольте DirectPlay заняться обработкой, а игроку продолжать игру, будто ничего не произошло.

## **Безопасность**

Страшно знать, что в любое время кто-то в сети может перехватывать и записывать ваши данные. Поэтому у вас есть возможность зашифровывать данные сообщений, что затрудняет хакерам чтение вашей драгоценной информации.

Обратной стороной безопасной доставки является замедление работы системы, поскольку данные надо зашифровать перед отправкой и расшифровать после приема. Если вы передаете важную информацию, потери времени не имеют значения (но они остаются важными для игр).

В DirectPlay есть встроенная поддержка безопасного обмена сообщениями. К счастью, для ее использования достаточно лишь установить соответствующий флаг в операции отправки. Теперь это бремя снято с ваших плеч.

## **Гарантированная доставка**

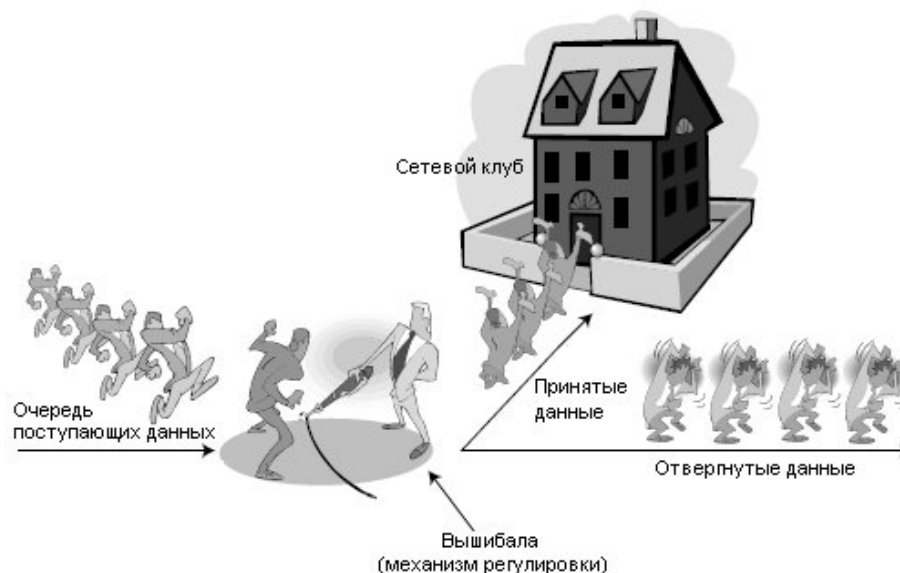
DirectPlay может гарантировать доставку сообщений, точно так же, как некоторые транспортные компании гарантируют доставку отправляемых через них посылок. Просто пометьте сообщение как требующее гарантированной доставки, и будьте уверены, что DirectPlay доставит его до места назначения (за исключением случая разрыва соединения), непрерывно повторяя операцию отправки до успешного завершения. Использование гарантированной доставки заключается в указании уникального флага в вызываемой функции, что вы увидите чуть позже в разделе «Отправка сообщений сервера» этой главы.

За гарантированную доставку придется платить скоростью. Гарантированная доставка слишком медленна, чтобы использовать ее в реальных игровых ситуациях. Игры используют метод доставки UDP (*User Datagram Protocol*), который не заботится о том, получены ли данные (в противоположность методу доставки TCP, гарантирующему доставку). Вы можете подумать, что это сумасшествие, но разобравшись в ситуации увидите, что игры очень часто отправляют обновленную информацию, и потеря небольшого количества данных время от времени вполне допустима.

## **Регулировка потока**

Иногда ваша система может испытывать перегрузки, пытаясь обработать поток сообщений. Однако, в DirectPlay есть встроенная система регулировки потока сообщений, которая выкидывает сообщения с низким приоритетом из очереди.

Возможно, рис. 5.6 поможет визуально представить концепцию использования механизма регулирования. Взглянув на рисунок, представьте очередь людей, ожидающих перед входом в популярный ночной клуб. Каждый человек представляет отдельное *сообщение*, и вышибала (*механизм регулирования*) должен удалить наименее важные из очереди, когда заведение становится переполненным.



*Рис. 5.6. Механизм регулировки (вышибала) принимает или отвергает сообщения (посетителей) в зависимости от их важности*

---

**ПРИМЕЧАНИЕ**

DirectPlay замечательно выполняет регулировку потока, так что обычно вам не требуется менять его параметры по умолчанию.

---

## От маленьких байтов к большим словам

Мир очень большой, все мы различны, и возникают ситуации, когда необходим посредник. Я говорю о языковом барьере, в частности о компьютерном языковом барьере.

DirectPlay введет вас в мир Unicode (если вы еще не знакомы с ним). *Unicode* — это универсальный стандарт, позволяющий различным программам и компьютерам совместно использовать информацию. Поскольку сеть может быть соединена с другой сетью, расположенной в любом месте мира, игроки могут жить в разных странах и говорить на разных языках.

Поскольку версии Windows в разных странах слегка отличаются, пользовательские системы можно настроить на использование символов Unicode. В результате вся система DirectPlay построена вокруг использования символов Unicode (для хранения каждого символа используется 16 бит вместо 8) и приспособлена к языкам, использующим более 255 символов. Я слышу ваш стон, но позвольте сказать, что в Windows есть все необходимые функции для преобразования из одного формата символов в другой, так что волноваться не о чем.

## Идентификация приложений по GUID

Сетевых приложений так много, как отличить свои от чужих? Присвойте вашему приложению уникальный номер и позвольте устанавливать соединение только с теми приложениями, у которых есть тот же номер. Этот

специальный номер, конечно же GUID (*Global Unique Identification*), знакомый Windows-программистам.

Создавая приложение, уделите минуту, чтобы назначить ему уникальный GUID и удостоверьтесь, что все приложения, которые будут соединяться с вашим по сети используют тот же самый GUID.

Вот и все, что я хотел сейчас сказать об отправке данных. Ход выполнения от создания сетевого объекта DirectPlay до отправки и получения данных для разных типов объектов очень похож, так что в последующем изложении я буду переплетать отдельную информацию.

## Инициализация сетевого объекта

Первый этап DirectPlay — создание сетевого объекта, будь это сервер, клиент или одноранговый пользователь. Для инициализации интерфейса каждой из этих сетевых моделей вы должны использовать функцию **CoCreateInstance** (которая детально описана в главе 4, «Воспроизведение звуков и музыки с DirectX Audio и DirectShow»), указав соответствующий идентификатор класса и интерфейса из представленных ниже:

CLSID_DirectPlay8Address	IID_IDirectPlay8Address
CLSID_DirectPlay8Client	IID_IDirectPlay8Client
CLSID_DirectPlay8Peer	IID_IDirectPlay8Peer
CLSID_DirectPlay8Server	IID_IDirectPlay8Server

Так, чтобы используя **CoCreateInstance** создать объект сервера, воспользуйтесь следующим фрагментом кода:

```
IDirectPlay8Server *pDPServer;

if (FAILED(CoCreateInstance(CLSID_DirectPlay8Server,
                           NULL, CLSCTX_INPROC,
                           IID_IDirectPlay8Server,
                           (void**) &pDPServer))) {
    // Произошла ошибка
}
```

### ВНИМАНИЕ!

Обязательно убедитесь, что инициализировали COM-систему перед вызовом **CoCreateInstance**. или любой функции DirectPlay. Для приложений, которые не используют несколько потоков (однопоточных), добавьте следующую строку до того, как делать какие-либо вызовы DirectX:

```
CoInitialize(NULL);
```

С другой стороны, если вы используете многопоточность, воспользуйтесь следующим вызовом функции:

```
CoInitializeEx(NULL, COINIT_MULTITHREADED);
```

Убедитесь, что при завершении работы ваше приложение вызывает функцию **CoUninitialize**:

```
CoUninitialize();
```

Независимо от того, какой сетевой объект вы создаете (клиент, одноранговый или сервер), необходимо создать соответствующую сетевую функцию обратного вызова. Эта функция вызывается каждый раз, когда принято сетевое сообщение. Вот как выглядит ее прототип:

```
typedef HRESULT (WINAPI *PFNDPNMESSAGEHANDLER) (  
    PVOID pvUserContext, // Предоставляемый приложением указатель  
    DWORD dwMessageType, // Тип полученного сообщения  
    PVOID pMessage);      // Буфер с зависящими от сообщения данными
```

Переменная **pvUserContext** — это указатель на какие-либо данные, которые вы хотите связать с игроком; этот указатель инициализируется при создании игрока. Он может быть указателем на структуру, хранящую состояние игры, или любую другую информацию по вашему желанию. Аргументы **dwMessageType** и **pMessage** относятся к сообщениям, и вы узнаете о них в разделе «Получение данных» далее в этой главе.

Сейчас можно пропустить создание самой функции обратного вызова и продолжить инициализацию сетевого объекта. Для завершения инициализации вызовите следующую функцию:

```
HRESULT IDirectPlay8Server::Initialize(  
    PVOID const pvUserContext, // Предоставляемый пользователем  
                                // указатель  
    const PFNDPNMESSAGEHANDLER pfn, // Функция обратного вызова  
                                // для обработки сообщений  
    const DWORD dwFlags);          // 0
```

---

<b>ПРИМЕЧАНИЕ</b>	Функция <b>Initialize</b> (с теми же самыми параметрами) работает в каждой из моделей:
-------------------	--

---

```
HRESULT IDirectPlay8Client::Initialize(...);  
HRESULT IDirectPlay8Peer::Initialize(...);
```

---

Вот пример создания вашей собственной функции обратного вызова (сейчас это только прототип) и инициализации только что созданного сетевого объекта сервера:

```
// Прототип функции обратного вызова  
HRESULT WINAPI MessageHandler(PVOID pvUserContext,  
                               DWORD dwMessageId, PVOID pMsgBuffer);  
// Инициализация ранее созданного объекта pDPsServer  
if(FAILED(pDPsServer->Initialize(NULL, MessageHandler, 0))) {  
    // Произошла ошибка  
}
```

Вот и все, что относится к созданию и инициализации сетевых объектов. Следующий этап — создание объекта адреса.

## Использование адресов

Как вы уже читали, в сети для доставки данных используется IP-адрес и номер порта. В DirectPlay вы конструируете этот адрес в его собственном объекте **IDirectPlay8Address**. Объект адреса предоставляет для

использования ряд функций, но в этой книге мы будем работать только с тремя из них (они перечислены в таблице 5.3).

**Таблица 5.3.** Функции IDirectPlay8Address

Функция	Описание
<b>IDirectPlay8Address::Clear</b>	Очищает все данные адреса.
<b>IDirectPlay8Address::SetSP</b>	Устанавливает поставщика услуг.
<b>IDirectPlay8Address::AddComponent</b>	Добавляет компоненты адреса.

## Инициализация объекта адреса

Перед тем, как вы сможете использовать объект адреса, необходимо создать его с помощью показанной ранее функции **CoCreateInstance**:

```
IDirectPlay8Address *pDPAAddress;

if (FAILED(CoCreateInstance(CLSID_DirectPlay8Address,
                           NULL, CLSCTX_INPROC,
                           IID_IDirectPlay8Address,
                           (void**) &pDPAAddress))) {
    // Произошла ошибка
}
```

## Добавление компонентов

Объект адреса просто содержит строку в формате Unicode. Эта строка содержит название поставщика услуг, номер порта, и другую информацию. Его единственное назначение — построить эту строку, которую будут использовать другие объекты.

Чтобы добавить компонент к объекту адреса используйте функцию **IDirectPlay8Address::AddComponent**:

```
HRESULT IDirectPlay8Address::AddComponent(
    const WCHAR *const pwszName, // Имя устанавливаемого компонента
    const void *const lpvData,   // Буфер, содержащий устанавливаемую
                                // информацию о компоненте
    const DWORD dwDataSize,      // Размер используемого буфера данных
    const DWORD dwDataType);     // Тип используемых данных
```

Эта небольшая функция требует большого количества пояснений, так что давайте приостановимся. Первый аргумент, **pwszName**, — это указатель на строку Unicode, содержащую имя добавляемого компонента. В DirectPlay есть макросы для каждого из этих компонентов, которые перечислены в таблице 5.4.

Какой буфер передавать функции **AddComponent** через аргумент **lpvData** зависит от типа передаваемого компонента, но обычно это строка (Unicode или ANSI), двойное слово (**DWORD**), GUID или двоичные данные.



Тип передаваемых данных сообщает параметр **dwDataType**, в котором можно использовать один из макросов, перечисленных в таблице 5.5.

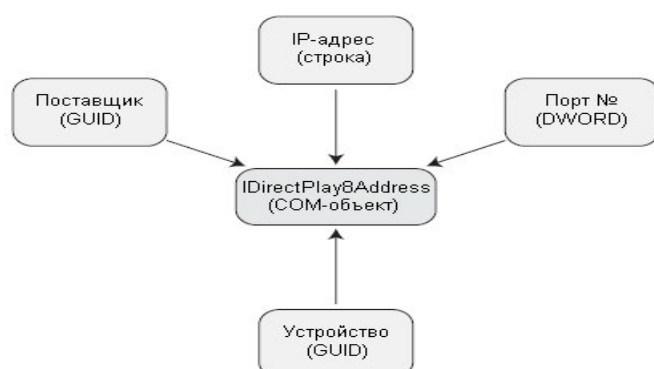
**Таблица 5.4.** Макросы для имен компонентов

<i>Компонент</i>	<i>Макрос</i>
Поставщик	DPNA_KEY_PROVIDER
Сетевое устройство	DPNA_KEY_DEVICE
Номер порта	DPNA_KEY_PORT
Имя/адрес узла	DPNA_KEY_HOSTNAME
Номер телефона	DPNA_KEY_PHONENUMBER
Скорость передачи	DPNA_KEY_BAUD
Контроль потока	DPNA_KEY_FLOWCONTROL
Четность	DPNA_KEY_PARITY
Стоп-биты	DPNA_KEY_STOPBITS

**Таблица 5.5.** Типы данных компонентов

<i>Тип</i>	<i>Макрос</i>
Строка (Unicode)	DPNA_DATATYPE_STRING
Строка (ANSI)	DPNA_DATATYPE_STRING_ANSI
DWORD	DPNA_DATATYPE_DWORD
GUID	DPNA_DATATYPE_GUID
Двоичные данные	DPNA_DATATYPE_BINARY

Последний аргумент, **dwDataSize**, определяет размер отправляемых данных (**DWORD**, длину строки, размер GUID и т.д.). Достаточно трудно представить использование этого метода для установки данных, но вам должны помочь примеры, которые я приведу в разделе «Выбор порта» далее в этой главе. Сейчас взгляните на рис. 5.7, где показаны компоненты, которые вы добавляете к объекту адреса, и тип данных каждого компонента.



**Рис. 5.7.** Объект адреса содержит информацию из различных компонентов. Каждому компоненту соответствует его тип данных

## Установка поставщика услуг

После того, как вы создали объект адреса, и поняли концепцию компонентов, следует выбрать тип поставщика услуг. Это единственный параметр, для установки которого не используется функция **AddComponent**. Вместо нее применяется функция **IDirectPlay8Address::SetSP**:

```
HRESULT IDirectPlay8Address::SetSP(
    const GUID *const pguidSP); // GUID поставщика услуг
```

GUID каждого поставщика услуг приведен в таблице 5.6. Выбор зависит от вас, но в этой книге я использую только поставщика услуг TCP/IP (**CLSID\_DP8SP\_TCPIP**).

**Таблица 5.6.** Поставщики услуг DirectPlay

<i>Тип</i>	<i>GUID</i>
TCP/IP	<b>CLSID_DP8SP_TCPIP</b>
IPX	<b>CLSID_DP8SP_IPX</b>
Модем	<b>CLSID_DP8SP_MODEM</b>
Последовательный порт	<b>CLSID_DP8SP_SERIAL</b>

С учетом вышесказанного, для установки используемого поставщика услуг можно использовать следующий код:

```
// Установка поставщика услуг TCP/IP
if(FAILED(pDPAddress->SetSP(&CLSID_DP8SP_TCPIP))) {
    // Произошла ошибка
}
```

## Выбор порта

Далее на очереди выбор порта, либо для открытия сессии (на сервере или в одноранговой сети), либо для подключения к удаленному компьютеру в сетевой модели клиента. Если вы подключаетесь к удаленной системе, то должны знать какой порт использует приложение, чтобы суметь установить соединение и отправить данные.

### ПРИМЕЧАНИЕ

Номер используемого порта можно выбирать, но не стоит использовать зарезервированные значения (от 1 до 1024). Для безопасности выберите какой-нибудь номер больше 1024.

Вы можете делегировать выбор порта DirectPlay, указав в номере порта 0, но тогда номер порта может быть любым и вам придется запрашивать его. Предпочтительнее самому выбирать порт.

Вы устанавливаете порт, используя функцию **IDirectPlay8Address::AddComponent**. Хотя, как я уже говорил,

эта функция может вызвать замешательство, вы вскоре увидите, как просто с ней работать.

Вот вызов, используемый для установки порта с помощью функции **AddComponent**:

```
// dwPort это значение типа DWORD представляющее
// номер используемого порта
if (FAILED(pDPAddress->AddComponent(DPNA_KEY_PORT, &dwPort,
                                     sizeof(DWORD), DPNA_DATATYPE_DWORD))) {
    // Произошла ошибка
}
```

Как видите, добавить компонент совсем не трудно, поскольку DirectPlay максимально облегчает задачу! На данный момент мы практически закончили инициализацию адреса. Что же осталось?

## Назначение устройства

Хотя вы и выбрали поставщика услуг, в вашей системе его могут использовать несколько устройств. Это происходит, например, в том случае, когда вы подключены к Интернету через сетевую карту и через модем — оба используют TCP/IP. В таких случаях вы должны перечислить устройства и выбрать необходимое.

### СОВЕТ

Если у вас только один адаптер, можете пропустить этот шаг, поскольку DirectPlay будет использовать первое найденное устройство.

На рис. 5.8 показан массив поставщиков услуг. Перечисление выбирает всех годных к использованию поставщиков услуг и предоставляет вам информацию о них в удобном для работы виде. Перечисление в DirectPlay слегка отличается от общепринятого в Windows метода перечисления; здесь не применяется функция обратного вызова, к которой обращаются каждый раз, когда найден экземпляр требуемого объекта. Вместо этого вы получаете буфер, содержащий данные всех поставщиков услуг в виде массива структур.



*Рис. 5.8. Перечисление создает список поставщиков услуг системы и помещает его в легкодоступный массив поставщиков*

Выполняется перечисление с помощью следующей функции (она также применима для сетевых объектов клиента и одноранговой сети):

```
HRESULT IDirectPlay8Server::EnumServiceProviders(
    const GUID *const pguidServiceProvider, // GUID поставщика услуг
    const GUID *const pguidApplication,     // NULL
    const DPN_SERVICE_PROVIDER_INFO *const pSPInfoBuffer,
    DWORD *const pcbEnumData,               // Указатель на DWORD
                                           // содержащий размер
                                           // буфера данных
    DWORD *const pcReturned,               // Указатель на DWORD
                                           // содержащий количество
                                           // перечисленных элементов
                                           // в буфере
    const DWORD dwFlags);                  // 0
```

Этой функции в первом параметре вы передаете GUID поставщика услуг (или **NULL**, если нужны все поставщики услуг). Параметр **pSPInfoBuffer** — это указатель на массив структур **DPN\_SERVICE\_PROVIDER\_INFO**, которые данная функция заполняет информацией, полученной при перечислении. Объявление этой структуры выглядит так:

```
typedef struct _DPN_SERVICE_PROVIDER {
    DWORD dwFlags; // 0
    GUID guid;     // GUID устройства
    WCHAR *pwszName; // Имя устройства
    PVOID pvReserved; // 0
    DWORD dwReserved; // 0
} DPN_SERVICE_PROVIDER;
```

Вызывая функцию вы предоставляете ей переменную типа **DWORD** (в параметре **pcbEnumData**), в которую записывается общий размер возвращаемых данных. Она нужна по единственной причине — выполнить предварительное перечисление, чтобы узнать требуемый размер буфера, и затем выделить достаточное количество памяти для хранения результатов перечисления.

Вот пример перечисления поставщиков услуг TCP/IP. Этот пример формирует список устройств, имеющих доступ к указанному поставщику услуг, и из этого списка вы можете выбрать устройство, которое будет использоваться в объекте адреса.

```
DPN_SERVICE_PROVIDER_INFO *pSP = NULL;
DPN_SERVICE_PROVIDER_INFO *pSPPtr;

DWORD dwSize = 0;
DWORD dwNumSP = 0;
DWORD i;

// Запрашиваем необходимый размер буфера данных
hr = pDPSServer->EnumServiceProviders(&CLSID_DP8SP_TCPIP,
                                       NULL, pSP, &dwSize, &dwNumSP, 0);

// Возвращаем код ошибки, если буфер мал,
// а все остальное нормально
```

```
if(hr != DPNERR_BUFFERTOOSMALL) {
    // Произошла непредусмотренная ошибка
} else {
    // Выделяем память под буфер и повторяем перечисление
    pSP = (DPN_SERVICE_PROVIDER_INFO*)new BYTE[dwSize];
    if(SUCCEEDED(pDPServer->EnumServiceProviders(
        &CLSID_DP8SP_TCPIP, NULL, pSP,
        &dwSize, &dwNumSP, 0))) {
        // Перечисление завершено, перебираем элементы
        pSPPtr = pSP;
        for(i = 0; i < dwNumSP; i++) {
            // pSPPtr->pwszName содержит строку Unicode
            // с именем поставщика
            // pSPPtr->guid содержит GUID поставщика услуг
            pSPPtr++; // Переходим к следующему
                    // поставщику услуг в буфере
        }
    }
    // Освобождаем занятую буфером память
    delete[] pSP;
}
```

---

**СОВЕТ**

Если вы хотите перечислить всех поставщиков услуг, а не только TCP/IP, замените `CLSID_DP8SP_TCPIP` на `NULL`.

---

Скорее всего, вы будете показывать названия поставщиков услуг пользователю, чтобы он выбрал того, который будет использовать. В комплекте DirectX SDK поставляется демонстрационная программа `AddressOverride`, показывающая как это делается.

Получив GUID поставщика услуг вы используете его для завершения формирования адреса. Это снова делается путем вызова функции **`IDirectPlay8Address::AddComponent`**, которой на этот раз передается GUID:

```
// guidSP = GUID поставщика услуг
if(FAILED(pDPAddress->AddComponent(DPNA_KEY_PROVIDER,
    &guidSP, sizeof(GUID), DPNA_DATATYPE_GUID))) {
    // Произошла ошибка
}
```

Теперь вы заполнили все компоненты адреса и можете использовать его.

## Использование обработчиков сообщений

Перед тем, как двигаться дальше, вы должны создать функцию обратного вызова для обработки сообщений. Логика этой функции прямолинейна. Она должна определить, какое сообщение было получено, и обработать его как можно быстрее. Можете представлять обработчик сообщений в виде воронки, изображенной на рис. 5.9.



*Рис. 5.9. Сетевой объект получает сообщения из сети и позволяет обработчику сообщений обрабатывать их одно за другим*

Прототип вы уже видели, так что взгляните на пример функции:

```
// Прототип функции обратного вызова
HRESULT WINAPI MessageHandler(PVOID pvUserContext,
                              DWORD dwMessageId,
                              PVOID pMsgBuffer)
{
    switch(dwMessageId) {
        case DPN_MSGID_CREATE_PLAYER:
            // Обработка создания игрока
            return S_OK;

        case DPN_MSGID_DESTROY_PLAYER:
            // Обработка удаления игрока
            return S_OK;
    }
    return E_FAIL;
}
```

Создав систему **switch...case** вы можете быстро отобрать необходимые сообщения, отбросив все остальные. Возврат значения **S\_OK** сообщает об успешной обработке сообщения. Значение **E\_FAIL** указывает, что возникла ошибка.

<b>ПРИМЕЧАНИЕ</b>	Также может быть возвращено значение <b>DPNSUCCESS_PENDING</b> , с которым вы будете иметь дело на стороне сервера. Подробнее о нем вы узнаете в разделе «Получение данных».
-------------------	--

---

Каждое сообщение поступает с буфером данных, который приводится к типу соответствующей структуры. Структуры следуют той же схеме именования, за исключением того, что их названия начинаются не с **DPN\_MSGID\_**, а с **DPNMSG\_**, как показано ниже:

```
DPNMSG_CREATE_PLAYER *pCreate; // DPN_MSGID_CREATE_PLAYER
DPNMSG_DESTROY_PLAYER *pDestroy; // DPN_MSGID_DESTROY_PLAYER
```

```
pCreate = (DPNMSG_CREATE_PLAYER*)pMsgBuffer;
pDestroy = (DPNMSG_DESTROY_PLAYER*)pMsgBuffer;
```

Конечно, содержание этих сообщений полезно только для конкретного сетевого объекта, так что нет необходимости использовать их все. Далее в этой главе, в разделах «Работа с сервером» и «Работа с клиентом» вы увидите использование сообщений с соответствующим объектом.

## Конфигурирование данных сессии

Каждый сетевой объект должен знать о сессии, которую собирается открыть или к которой хочет присоединиться. Эта информация хранится в отдельной структуре:

```
typedef struct _DPN_APPLICATION_DESC {
    DWORD dwSize;           // Размер структуры
    DWORD dwFlags;          // Флаги сессии
    GUID guidInstance;      // NULL
    GUID guidApplication;   // GUID приложения
    DWORD dwMaxPlayers;     // Максимально доступное
                          // количество игроков
    DWORD dwCurrentPlayers; // Текущее количество игроков
    WCHAR *pwszSessionName; // Имя сессии
    WCHAR *pwszPassword;    // Пароль сессии (если нужен)
    PVOID pvReservedData;   // Не используется
    DWORD dwReservedDataSize; // Не используется
    PVOID pvApplicationReservedData; // NULL
    DWORD dwApplicationReservedDataSize; // 0
} DPN_APPLICATION_DESC;
```

Как я уже сказал, вся информация из структуры **DPN\_APPLICATION\_DESC** не требуется, так что мы отдельно обсудим, что надо для каждой сетевой модели.

## Данные сессии сервера

Для сервера необходимо задать максимально допустимое количество игроков (если вы хотите установить ограничение), имя сессии и пароль, используемые при регистрации, флаги сессии и GUID приложения.

Чтобы сделать это, очистите структуру (заполнив ее нулями), установите переменную **dwSize**, и заполните требуемые поля. Для переменной **dwFlags** вы можете выбрать один из флагов, перечисленных в таблице 5.7.

**Таблица 5.7.** Флаги сессии

<i><b>Макрос флага</b></i>	<i><b>Описание</b></i>
<b>DPNSESSION_CLIENT_SERVER</b>	Это сессия модели клиент/сервер.
<b>DPNSESSION_MIGRATE_HOST</b>	Используется в одноранговой модели, включение этого флага заставляет DirectPlay переносить информацию узла на другую систему в случае потери текущего узла.
<b>DPNSESSION_NODPNSVR</b>	Запрещает DirectPlay выполнять перечисление вашего приложения с удаленной системы.
<b>DPNSESSION_REQUIREPASSWORD</b>	Для подключения удаленная система должна указать правильный пароль.
<hr/>	
<b>ПРИМЕЧАНИЕ</b>	Если вы не хотите задавать максимально возможное количество игроков, оставьте в этом поле 0. То же самое с паролем; просто не используйте флаг и оставьте поле пароля в покое.

Вот пример установки информации сессии:

```
// GUID приложения
GUID AppGUID = { 0xede9493e, 0x6ac8, 0x4f15,
                 { 0x8d, 0x1, 0x8b, 0x16, 0x32, 0x0, 0xb9, 0x66 } };

// Обнуляем структуру данных сессии и устанавливаем ее размер
DPN_APPLICATION_DESC dpad;

ZeroMemory(&dpad, sizeof(DPN_APPLICATION_DESC));
dpad.dwSize = sizeof(DPN_APPLICATION_DESC);

// Задаем имя и пароль сессии
dpad.pwszSessionName = L"MySession";
dpad.pwszPassword    = L"MyPassword";

// Устанавливаем максимальное количество игроков равным 4
dpad.dwMaximumPlayers = 4;

// Устанавливаем GUID приложения
dpad.guidApplication = AppGUID;

// Устанавливаем флаги модели клиент/сервер и использования пароля
dpad.dwFlags = DPNSESSION_CLIENT_SERVER |
               DPNSESSION_REQUIREPASSWORD;
```



## Данные сессии клиента

Информация, которую надо установить в структуре данных сессии клиента включает имя и пароль сессии, к которой вы хотите подключиться, флаг модели клиент/сервер и GUID приложения. Убедитесь, что указали тот же GUID, что и для приложения сервера, чтобы они могли найти друг друга в сети. А вот пример:

```
// GUID приложения сервера
GUID AppGUID = { 0xede9493e, 0x6ac8, 0x4f15,
                 { 0x8d, 0x1, 0x8b, 0x16, 0x32, 0x0, 0xb9, 0x66 } };

// Обнуляем структуру данных сессии и устанавливаем ее размер
DPN_APPLICATION_DESC dpad;

ZeroMemory(&dpad, sizeof(DPN_APPLICATION_DESC));
dpad.dwSize = sizeof(DPN_APPLICATION_DESC);

// Задаем имя и пароль сессии к которой подключаемся
dpad.pwszSessionName = L"MySession";
dpad.pwszPassword     = L"MyPassword";

// Устанавливаем GUID приложения
dpad.guidApplication = AppGUID;

// Устанавливаем флаги модели клиент/сервер и использования пароля
dpad.dwFlags = DPNSSESSION_CLIENT_SERVER |
               DPNSSESSION_REQUIREPASSWORD;
```

## Работа с сервером

Наконец-то начинается настоящее развлечение! Первый этап реальной работы с сетью — это создание сервера. Сервер действует как центральный процессор вашей сетевой игры. Все игроки подключаются к серверу через клиентское приложение и начинаются передача данных туда-сюда.

Сервер поддерживает синхронизацию данных и оповещает игроков о текущем состоянии игры. Хотя для небольших сетевых игр это, возможно, не самый быстрый метод, для крупномасштабных игр он является наилучшим, и поэтому я использую его в главе 15.

Вот завершенная функция, которая создает объект сервера и инициализирует его, устанавливая функцию обработки сообщений (пока очень простую), создает компоненты адреса и структуру данных сессии, а затем выполняет специальный вызов для открытия игровой сессии. В результате вызова этой функции вы получите указатель на сетевой объект сервера.

```
// GUID сервера
GUID AppGUID = { 0xede9493e, 0x6ac8, 0x4f15,
                 { 0x8d, 0x1, 0x8b, 0x16, 0x32, 0x0, 0xb9, 0x66 } };

// Прототип обработчика сообщений
HRESULT WINAPI ServerMsgHandler(PVOID pvUserContext,
                                DWORD dwMessageId, PVOID pMsgBuffer);
```

```

IDirectPlay8Server *StartNetworkServer(
    char    *szSessionName, // Имя сессии (в ANSI)
    char    *szPassword,    // Используемый пароль
                        // (NULL если нет)
    DWORD   dwPort,         // Используемый порт
    DWORD   dwMaxPlayers)   // Максимальное количество
                        // игроков
{
    IDirectPlay8Server *pDPServer;
    IDirectPlay8Address *pDPAddress;

    DPN_APPLICATION_DESC dpad;

    WCHAR wszSessionName[256];
    WCHAR wszPassword[256];

    // Создание и инициализация объекта сервера
    if (FAILED(DirectPlay8Create(&IID_IDirectPlay8Server,
                                (void**)&pDPServer, NULL)))
        return NULL;
    if (FAILED(pDPServer->Initialize(pDPServer,
                                    ServerMsgHandler, 0))) {
        pDPServer->Release();
        return NULL;
    }

    // Создание объекта адреса, установка поставщика услуг
    // и порта
    if (FAILED(DirectPlay8AddressCreate(
        &IID_IDirectPlay8Address, (void**)&pDPAddress, NULL))) {
        pDPServer->Release();
        return NULL;
    }
    pDPAddress->SetSP(&CLDID_DP8SP_TCPIP);
    pDPAddress->AddComponent(DPNA_KEY_PORT, &dwPort,
                            sizeof(DWORD), DPNA_DATATYPE_DWORD);

    // Установка данных сессии
    ZeroMemory(&dpad, sizeof(DPNA_APPLICATION_DESC));
    dpad.dwSize = sizeof(DPNA_APPLICATION_DESC);
    dpad.dwFlags = DPNSESSION_CLIENT_SERVER;

    // Установка имени сессии с преобразованием ANSI в Unicode
    mbstowcs(wszSessionName, szSessionName,
              strlen(szSessionName) + 1);
    dpad.pwszSessionName = wszSessionName;

    // Установка пароля (если надо)
    if (szPassword != NULL) {
        mbstowcs(wszPassword, szPassword, strlen(szPassword)+1);
        dpad.pwszPassword = wszPassword;
        dpad.dwFlags |= DPNSESSION_REQUIREPASSWORD;
    }

    // Установка максимального количества игроков
    dpad.dwMaxPlayers = dwMaxPlayers;
}

```

Здесь я на секунду останавлиюсь и познакомлю вас со специальной функцией узла, которая открывает сетевую сессию объекта сервера:

```
HRESULT IDirectPlay8Server::Host(
    const DPN_APPLICATION_DESC *const pdnAppDesc,          // Данные сессии
    IDirectPlay8Address **const prgpDeviceInfo,            // Объект адреса
    const DWORD cDeviceInfo,                                // 1
    const DPN_SECURITY_DESC *const pdpSecurity,            // NULL
    const DPN_SECURITY_CREDENTIALS *const pdpCredentials,  // NULL
    VOID *const dwPlayerContext,                            // NULL
    const DWORD dwFlags);                                   // 0
```

К счастью, для этой монстрообразной функции уже все готово. Вы уже создали объект адреса и инициализировали описание сессии, а все остальное — детские игрушки. Вот оставшаяся часть функции:

```
    if (FAILED(pDPServer->Host(&dpad, &pDPAddress, 1,
                               NULL, NULL, NULL, 0))) {
        pDPAddress->Release();
        pDPServer->Release();
        return NULL;
    }

    // Освобождаем объект адреса - он больше не требуется
    pDPAddress->Release();

    // Возвращаем объект сервера
    return pDPServer;
}

// Функция серверного обработчика сообщений с размещенной на месте
// и готовой к употреблению конструкцией switch...case для сообщений
HRESULT WINAPI ServerMsgHandler(PVOID pvUserContext,
                                DWORD dwMessageId, PVOID pMsgBuffer)
{
    // Определите здесь структуры сообщений DPNMSG_*

    // Указатель на вызывающий объект сервера передается через
    // пользовательский указатель при вызове Initialize

    IDirectPlay8Server *pDPServer;
    pDPServer = (IDirectPlay8Server*)pvUserContext;

    switch(dwMessageId) {
        // Добавьте здесь инструкции case для различных
        // типов сообщений, например:
        // case DPN_MSGID_CREATE_PLAYER:
        //     DPNMSG_CREATE_PLAYER dpcp;
        //     dpcp = (DPNMSG_CREATE_PLAYER*)pMsgBuffer;
        //     Делайте, что вам надо с этими данными и, когда закончите,
        //     верните флаг успеха
        //     return S_OK;
    }
    return E_FAIL;
}
```

Как видите, я снова вставил только скелет функции обработчика сообщений. Перед тем, как вы сможете начать работу с сообщениями, надо понять лежащую в их основе теорию. В следующих разделах мы начнем с относящихся к игрокам сообщений, поскольку они наиболее часто используются.

## Поддержка игроков

При запуске сервера одним из первых вы получите сообщение о создании игрока. Первый игрок всегда создается для главного узла. Другие игроки могут приходить и уходить, а игрок главного узла остается в течение всей сессии.

### Работа с сообщениями о создании игрока

Сообщение о создании игрока определено как **DPN\_MSGID\_CREATE\_PLAYER**, и буфер сообщения приводится к типу структуры **DPNMSG\_CREATE\_PLAYER**, объявление которой выглядит так:

```
typedef struct _DPNMSG_CREATE_PLAYER {
    DWORD dwSize;           // Размер структуры
    DWORD dpnidPlayer;      // Идентификатор игрока
    PVOID pvPlayerContext;  // Указатель на данные контекста игрока
} DPNMSG_CREATE_PLAYER;
```

Изумительно простая структура содержит лишь два полезных фрагмента информации: назначенный игроку идентификатор, который в дальнейшем вы будете использовать для ссылки на игрока, и указатель на данные контекста игрока.

#### ПРИМЕЧАНИЕ

*Контекст игрока (player context)* — это информация, которую вы используете для описания игрока в вашем приложении. Это может быть указатель на структуру, экземпляр класса или буфер данных, который содержит имя игрока, уровень здоровья, возраст, текущее оружие, броню и т.д.

Предоставляя DirectPlay указатель на этот контекст игрока, вы обеспечиваете DirectPlay возможность в дальнейшем предоставить вам способ быстрого доступа к данным. Это избавляет от необходимости перебирать весь список игроков, ища требуемый идентификатор, когда время поджимает!

Как видите, в структуре **DPNMSG\_CREATE\_PLAYER** ряд данных отсутствует, в частности, имя игрока. Это работа отдельной функции, с которой вы познакомитесь в следующем разделе, «Получение имени игрока». Сейчас вы устанавливаете контекст игрока, что осуществляется простым приведением типа указателя:

```
DPNMSG_CREATE_PLAYER pCreatePlayer;
pCreatePlayer->pvPlayerContext = (PVOID)ContextDataPtr;
```

Конечно, **ContextDataPtr** — это указатель на что-то, что вы используете для хранения данных игрока. Чтобы связать единственную структуру из массива структур, содержащих внутриигровую информацию об игроках, передайте в виде контекста указатель на структуру, как показано в следующем примере:

```
typedef struct {
    char  szPlayerName[32]; // Имя игрока
    DWORD dwXPos, dwYPos;   // Координаты игрока
} sPlayerInfo;

sPlayerInfo Players[100]; // Место для 100 игроков

// Внутри обрабатывающей сообщения конструкции
// switch...case в секции создания игрока:
pCreatePlayer->pvPlayerContext = (PVOID)&sPlayerInfo[1];
```

### **Получение имени игрока**

У игрока есть связанное с ним имя и вы должны суметь получить эту информацию, чтобы использовать в игре (кто хочет, чтобы его называли по номеру?). Это работа функции **IDirectPlay8Server::GetClientInfo**:

```
HRESULT IDirectPlay8Server::GetClientInfo(
    const DPNID dpnid,           // Идентификатор игрока
    DPN_PLAYER_INFO *const pdpnPlayerInfo, // Структура данных игрока
    DWORD *const pdwSize,       // Размер предыдущей
                                // структуры
    const DWORD dFlags);        // 0
```

И снова вы имеете дело со структурой данных, которая может быть любого размера, поэтому вам сперва надо запросить правильный размер, выделить буфер, а затем получить структуру. Буфер данных — это форма структуры **DPN\_PLAYER\_INFO**, показанной ниже:

```
typedef struct _DPN_PLAYER_INFO {
    DWORD dwSize;           // Размер структуры
    DWORD dwInfoFlags;      // DPNINFO_NAME | DPNINFO_DATA
    PWSTR pwszName;         // Имя игрока (в Unicode)
    PVOID pvData;           // Указатель на данные игрока
    DWORD dwDataSize;       // Размер данных игрока
    DWORD dwPlayerFlags;    // DPNPLAYER_LOCAL для локального игрока
                          // или DPNPLAYER_HOST для игрока главного узла
} DPN_PLAYER_INFO;
```

Вы посмотрели на магические параметры, а теперь давайте пойдем дальше и взглянем на процесс обработки сообщения о создании игрока и извлечения связанного с игроком имени:

```
HRESULT WINAPI ServerMsgHandler(PVOID pvUserContext,
                                DWORD dwMessageId, PVOID pMsgBuffer)
{
    IDirectPlay8Server *pDPSServer;
    HRESULT hr;
    DPNMSG_CREATE_PLAYER *pCreatePlayer;
    DPN_PLAYER_INFO *dppli;
    DWORD dwSize;

    if((pDPSServer = (IDirectPlay8Server*)pvUserContext) == NULL)
        return E_FAIL;

    switch(dwMessageId) {
        case DPN_MSGID_CREATE_PLAYER:
```

```

pCreatePlayer = (DPNMSG_CREATE_PLAYER*)pMsgBuffer;
dwSize = 0;
dppi = NULL;

// Запрашиваем размер буфера данных
hr = pDPSServer->GetClientInfo(
    pCreatePlayer->dpnidPlayer, dppi, &dwSize, 0);

// Проверка ошибок - если это недопустимый игрок,
// значит добавляется игрок главного узла (пропускаем его)
if(FAILED(hr) && hr != DPNERR_BUFFERTOOSMALL) {
    if(hr == DPNERR_INVALIDPLAYER)
        break;
}

// Выделяем буфер данных и получаем информацию
dppi = (DPN_PLAYER_INFO*)new BYTE[dwSize];
ZeroMemory(dppi, sizeof(DPN_PLAYER_INFO));
dppi->dwSize = sizeof(DPN_PLAYER_INFO);
if(FAILED(pDPSServer->GetClientInfo(
    pCreatePlayer->dpnidPlayer, dppi, &dwSize, 0))) {
    delete[] dppi;
    break;
}

// Теперь у нас есть информация игрока в структуре dppi
// Чтобы получить имя игрока в кодировке ANSI, обратитесь
// к функции wcstombs

// Сейчас мы просто отображаем имя в окне сообщений
char szName[32];
wcstombs(szName, dppi->pwszName, 32);
MessageBox(NULL, szName, "Player Joined", MB_OK);

// Избавляемся от буфера с данными игрока
delete[] dppi;
return S_OK;
}
return E_FAIL;
}

```

**ВНИМАНИЕ!**

Обратите внимание, что запрос получения данных игрока может закончиться неудачей не по вашей вине. Клиент может не установить структуру с информацией клиента и, следовательно, сервер не сможет получить ее.

**Уничтожение игроков**

Нет, вы не убиваете их в игре, но когда игрок разрывает соединение, вы получаете сообщение об этом. Создать сообщение также просто, как и в случае создания игрока. Буфер сообщения необходимо привести к типу структуры **DPNMSG\_DESTROY\_PLAYER**:

```

typedef struct _DPNMSG_DESTROY_PLAYER {
    DWORD dwSize;           // Размер структуры
    DPNID dpnidPlayer;       // Идентификатор удаляемого игрока
    PVOID pvPlayerContext;   // Указатель на контекст игрока
    DWORD dwReason;          // Причина отключения
} DPNMSG_DESTROY_PLAYER;

```

Вы снова используете идентификатор игрока и указатель на его контекст, которые уже видели ранее. Вопросы вызывает только последнее поле, **dwReason**. Почему игрок покинул игру? Было ли это обычное завершение игры, или было внезапно разорвано соединение, может была прервана сессия или игрок был принудительно отключен? Каждой из этих причин соответствует макрос из перечисленных в таблице 5.8. Вы можете использовать значение в поле **dwReason** как считаете нужным.

**Таблица 5.8.** Причины отключения

<i>Макрос</i>	<i>Описание</i>
<b>DPNDESTROYPLAYERREASON_NORMAL</b>	Обычное отключение игрока.
<b>DPNDESTROYPLAYERREASON_CONNECTIONLOST</b>	Отключение игрока из-за разрыва соединения.
<b>DPNDESTROYPLAYERREASON_SESSIONTERMINATED</b>	Удаление игрока из-за завершения сессии.
<b>DPNDESTROYPLAYERREASON_HOSTDESTROYPLAYER</b>	Принудительное удаление игрока сервером.

Для принудительного отключения игрока вы используете функцию **IDirectPlay8Server::DestroyClient**:

```
HRESULT IDirectPlay8Server::DestroyClient(  
    const DPNID pdnidClient,          // Идентификатор игрока  
    const void *const pDestroyInfo,  // NULL  
    const DWORD dwDestroyInfoSize,   // 0  
    const DWORD dwFlags);            // 0
```

Здесь нет ничего нового — просто укажите идентификатор игрока. Вот как выполняется отключение:

```
pDPSTServer->DestroyClient(dpnidPlayerID, NULL, 0, 0);
```

Другой способ отключения игрока — завершение сессии. Об этом мы поговорим чуть позже в разделе «Завершение сессии на главном узле».

## Получение данных

Игровые данные передаются в виде зависящих от приложения сообщений, но они всегда заключены в сообщения типа **DPN\_MSGID\_RECEIVE**, использующих структуры данных **DPNMSG\_RECEIVE**:

```
typedef struct _DPNMSG_RECEIVE {  
    DWORD dwSize;                // Размер структуры  
    DPNID dpnidSender;           // Идентификатор отправителя  
    PVOID pvPlayerContext;       // Указатель на контекст игрока  
    PBYTE pReceiveData;          // Буфер принятых данных  
    DWORD dwReceivedDataSize;    // Размер принятых данных  
    DPNHANDLE hBufferHandle;     // Дескриптор буфера данных  
} DPNMSG_RECEIVE;
```

Чтобы обработать данные, обращайтесь к ним через указатель **pReceiveData**, используя, если необходимо, дескриптор памяти Windows **hBufferHandle**. Сперва это кажется бессмысленным, но в действительности гарантирует, что сообщение будет храниться в памяти, пока вы не будете готовы работать с данными.

В качестве примера предположим, что будет принято 16 байт данных. Эти данные представляют состояние игрока в игре. Для доступа к данным выполните приведение типа к указателю на структуру данных и работайте с данными, как показано в следующем примере:

```
#define MSG_PLAYERSTATE 0x101

typedef struct {
    DWORD dwType;
    DWORD dwXPos, dwYPos, dwHealth;
} sPlayerState;

// Обработчик сообщений
HRESULT WINAPI ServerMsgHandler(PVOID pvUserContext,
                                DWORD dwMessageId, PVOID pMsgBuffer)
{
    IDirectPlay8Server *pDPSServer;
    HRESULT hr;
    DPNMSG_RECEIVE *pReceive;
    sPlayerState *pState;

    if((pDPSServer = (IDirectPlay8Server*)pvUserContext) == NULL)
        return E_FAIL;

    switch(dwMessageId) {
        case DPN_MSGID_RECEIVE:
            pReceive = (DPNMSG_RECEIVE*)pMsgBuffer;

            // Приведение буфера данных к типу сообщения
            pState = (sPlayerState*)pReceive->pReceivedData;
            if(pState->dwType == MSG_PLAYERSTATE) {

                // Делаем что нам надо со структурой данных

            }
            return S_OK;
        }
    return E_FAIL;
}
```

Иногда поступает очень много сообщений и вы не можете обработать их все при получении. В таком случае можно поместить их в очередь. Когда вы закончите работать с данными в памяти, передайте дескриптор функции **IDirectPlay8Server::ReturnBuffer**:

```
HRESULT IDirectPlay8Server::ReturnBuffer(
    const DPNHANDLE hBufferHandle, // Дескриптор буфера
    const DWORD dwFlags);         // 0
```



Чтобы DirectPlay знал, что освобождать память не надо, после завершения обработки сообщения верните значение **DPNSUCCESS\_PENDING**, вместо **S\_OK** или **E\_FAIL**.

## Отправка сообщений сервера

Что хорошего в сети, которая не может передавать данные? Чтобы сервер отправил данные подключенному клиенту вам надо использовать функцию **SendTo**, которая отправляет данные отдельному игроку, всем игрокам сразу или игрокам, относящимся к указанной группе. Взгляните на прототип функции **SendTo**:

```
HRESULT IDirectPlay8Server::SendTo(
    const DPNID dpnid,                // Идентификатор игрока или группы,
                                      // которым отправляется сообщение
                                      // Для отправки сообщения всем
                                      // игрокам используйте
                                      // DPNID_ALL_PLAYERS_GROUP
    const DPN_BUFFER_DESC *const pBufferDesc, // См. описание
    const DWORD cBufferDesc,            // 1
    const DWORD dwTimeOut,              // Время ожидания отправки
                                      // сообщения (в миллисекундах)
                                      // 0 - если время ожидания
                                      // не задано
    void *const pvAsyncContext,         // Предоставленный пользователем
                                      // контекст
    DPNHANDLE *const phAsyncHandle,     // NULL для синхронной операции
    const DWORD dwFlags);              // См. описание
```

Функция **SendTo** выглядит подавляюще. Вам необходимо учесть безопасность, метод доставки и регулировку потока. Вы должны указать идентификатор игрока, которому хотите отправить сообщение, а также указатель на структуру **DPN\_BUFFER\_DESC**. Определение этой простой структуры выглядит так:

```
typedef struct _DPN_BUFFER_DESC {
    DWORD dwBufferSize; // Размер передаваемых данных
    BYTE *pBufferData;  // Указатель на передаваемые данные
} DPN_BUFFER_DESC;
```

Здесь вы задаете размер и указатель на данные, которые хотите отправить. Затем в функции **SendTo** идет параметр **cBufferDesc**, которому присваивается значение 1. Далее в **dwTimeOut** устанавливается значение, определяющее период времени (в миллисекундах), который функция будет ждать, прежде чем вернет ошибку (со времени отправки данных). Если вы не хотите использовать эту возможность, просто укажите в данном параметре 0.

Аргумент **pvAsyncContext** — это задаваемый пользователем контекст, используемый для указания на информацию, которую вы хотите получить, когда операция отправки будет завершена. Он похож на контекст игрока, поскольку упрощает доступ к информации.

Для использования асинхронной отправки вы предоставляете аргумент **phAsyncHandle**, куда будет записан дескриптор, который позже можно использовать для отмены операции отправки. Осталось рассмотреть **dwFlags**. Взгляните на таблицу 5.9, где приведен список макросов, которые можно использовать для конструирования этого значения, с их описанием.

**Таблица 5.9.** Флаги поведения SendTo

<i><b>Макрос</b></i>	<i><b>Описание</b></i>
<b>DPNSSEND_SYNC</b>	Синхронная отправка данных. Не возвращаем управление, пока данные не отправлены.
<b>DPNSSEND_NOCOPY</b>	Заставляет DirectPlay не делать внутреннюю копию отправляемых данных. Это самый эффективный метод отправки данных; однако отложенные данные могут быть изменены прежде чем DirectPlay получит шанс отправить их.
<b>DPNSSEND_NOCOMPLETE</b>	Указывает DirectPlay, что не надо уведомлять сервер, когда операция отправки завершена.
<b>DPSSEND_COMPLETEONPROCESS</b>	Заставляет DirectPlay отправлять сообщение <b>DPN_MSGID_SEND_COMPLETE</b> , когда данные отправлены и проверены системой на месте назначения. Это замедляет работу, но позволяет гарантировать, что данные доставлены. Вместе с этим флагом вы должны указывать флаг <b>DPSSEND_GUARANTEED</b> .
<b>DPSSEND_GUARANTEED</b>	Использование гарантированной доставки.
<b>DPNSSEND_PRIORITY_HIGH</b>	Задает высокий приоритет для сообщения. Используется для отметки важных сообщений, которые должны быть пропущены через механизм фильтрации.
<b>DPNSSEND_PRIORITY_LOW</b>	Задает низкий приоритет для сообщения. Используйте этот флаг для отметки не слишком важных сообщений, которые могут быть отброшены механизмом фильтрации.
<b>DPNSSEND_NOLOOPBACK</b>	Подавляет сообщение <b>DPN_MSGID_RECEIVE</b> на сервере, если вы отправляете данные группе в которую входит игрок сервера.
<b>DPNSSEND_NONSEQUENTIAL</b>	Заставляет систему назначения принимать сообщения в том порядке, в котором они были отправлены (а не быть перепутанными из-за задержек в сети). Непоследовательный прием может замедлить работу удаленной стороны, поскольку потребуется отслеживать, буферизовать и переупорядочивать сетевые сообщения, чтобы они были доставлены в том порядке в котором их отправляли.

Типичная комбинация флагов — **DPNSEND\_NOLOOPBACK** и **DPNSEND\_NOCOPY** — обеспечивает оптимальную производительность без вовлечения игрока сервера в отправляемые группам сообщения.

Вернемся к примеру получения сообщения. Вы можете сделать свой ход и отправить информацию тому же игроку, возможно слегка изменив данные.

Вот снова инструкция **switch...case**, но теперь с отправкой информации:

```
DPNHANDLE g_hSendTo; // Асинхронный дескриптор для данных

switch(dwMessageId) {
    case DPN_MSGID_RECEIVE:
        pReceive = (DPNMSG_RECEIVE*)pMsgBuffer;

        // Приведение буфера данных к типу сообщения
        pState = (sPlayerState*)pReceive->pReceivedData;

        if(pState->dwType == MSG_PLAYERSTATE) {
            // Изменение данных
            pState->dwHealth += 10; // Увеличиваем здоровье

            // Приведение типа для отправки
            DPN_BUFFER_DESC dpbd;
            dpbd.dwSize = sizeof(sPlayerState);
            dpbd.pBufferData = pState;

            // Отправка с использованием внутреннего
            // метода копирования и без отправки уведомления
            // Возвращает асинхронный дескриптор для остановки
            // операции отправки
            pDPServer->SendTo(pReceive->dpnidSender, &dpbd,
                            1, 0, NULL, &g_hSendTo, DPNSEND_NOCOMPLETE);
        }
        return S_OK;
}
```

Чтобы удалить информацию до ее отправки (пока она ожидает в очереди отправки), используйте глобальный дескриптор следующим образом:

```
// Для отмены одной операции отправки используйте:
pDPServer->CancelAsyncOperation(g_hSendTo, 0);

// Для отмены всех отложенных операций используйте:
pDPServer->CancelAsyncOperation(NULL, DPNCANCEL_ALL_OPERATIONS);
```

## Завершение сессии на главном узле

Когда сервер завершает работу, наступает время для остановки сессии, что прекращает все передачи и уничтожает всех игроков. Это делается с помощью следующей функции:

```
HRESULT IDirectPlay8Server::Close(
    const DWORD dwFlags); // 0
```

---

**ПРИМЕЧАНИЕ**      Функция `close` работает с сетевыми объектами всех моделей, так что вы не ограничены только сервером.

---

Поскольку эта функция работает синхронно, она не возвратит управление, пока не будут завершены все операции передачи и не будут закрыты все соединения. Это гарантирует, что выключение приложения не вызовет проблем.

И, наконец, освободите все используемые COM-объекты; в данном случае это только объект сервера:

```
pDPSServer->Release();
```

## Работа с клиентом

Клиент, в свою очередь, не так сложен как сервер. Он обычно использует только два сообщения, о приеме данных и о прекращении сессии, и устанавливает и отслеживает единственное соединение (с сервером).

Основное дополнение состоит в том, что клиентское приложение должно установить параметры игрока, чтобы главный узел мог получить их. Установка информации об игроке заключается в заполнении соответствующими данными структуры **DPN\_PLAYER\_INFO** и последующем вызове функции **IDirectPlay8Client::SetClientInfo**.

Вам интересны лишь несколько полей структуры **DPN\_PLAYER\_INFO** — особенно **pwszName**, содержащее строку Unicode с именем игрока, которое вы хотите использовать. Вам необходимо очистить структуру, установить значение **dwSize**, присвоить полю **dwInfoFlags** значение **DPNINFO\_NAME | DPNINFO\_DATA** и задать имя игрока.

Вот прототип функции **IDirectPlay8Client::SetClientInfo**:

```
HRESULT IDirectPlay8Client::SetClientInfo(
    const DPN_PLAYER_INFO *const pdpnPlayerInfo, // Данные игрока
    PVOID const pvAsyncContext,                  // NULL
    DPNHANDLE *const phAsyncHandle,               // NULL
    const DWORD dwFlags);                        // DPNSETCLIENTINFO_SYNC
```

Здесь вы снова видите указатель на структуру с информацией об игроке. Ниже приведена законченная функция, которую вы можете использовать для создания объекта клиента, его инициализации с указанием обработчика сообщений, создания объекта адреса, инициализации сессии и данных о клиенте и подключения к серверу. Мы не будем обсуждать функцию обработки сообщений; ее назначение то же самое, что и на сервере.

---

**ВНИМАНИЕ!**      Как уже упоминалось, используйте один и тот же GUID приложения для клиента и для сервера, чтобы они могли распознать друг друга. Невыполнение этого требования — одна из главных причин, по которой сетевые приложения не могут установить соединение, так что убедитесь, что GUID приложений одинаковы.

---

```
// GUID клиента/сервера
GUID AppGUID = { 0xede9493e, 0x6ac8, 0x4f15,
                 { 0x8d, 0x1, 0x8b, 0x16, 0x32, 0x0, 0xb9, 0x66 } };

// Прототип обработчика сообщений
HRESULT WINAPI ClientMsgHandler(PVOID pvUserContext,
                                DWORD dwMessageId, PVOID pMsgBuffer);

IDirectPlay8Client *StartClientServer(
    char *szPlayerName, // Имя игрока
    char *szSessionName, // Имя сессии, к которой
                        // подключаемся (в ANSI)
    char *szPassword, // Используемый пароль (NULL если не нужен)
    char *szIPAddress, // Текстовая строка с IP-адресом
                        // в формате ###.###.###.###
    DWORD dwPort) // Используемый порт
{
    IDirectPlay8Client *pDPClient;
    IDirectPlay8Address *pDPAddress;

    DPN_APPLICATION_DESC dpad;
    DPN_PLAYER_INFO dppl;

    WCHAR wszSessionName[256];
    WCHAR wszPassword[256];
    WCHAR wszIPAddress[256];
    WCHAR wszPlayerName[256];

    // Создание и инициализация объекта клиента
    if (FAILED(DirectPlay8Create(&IID_IDirectPlay8Client,
                                (void**) &pDPClient, NULL)))
        return NULL;
    if (FAILED(pDPClient->Initialize(pDPClient,
                                    ClientMsgHandler, 0))) {
        pDPClient->Release();
        return NULL;
    }

    // Создание объекта адреса и установка поставщика услуг
    if (FAILED(DirectPlay8AddressCreate(&IID_IDirectPlay8Address,
                                        (void**) &pDPAddress, NULL))) {
        pDPClient->Release();
        return NULL;
    }
    pDPAddress->SetSP(&CLDID_DP8SP_TCPIP);

    // Преобразование IP-адреса в Unicode и добавление компонента
    mbstowcs(wszIPAddress, szIPAddress, strlen(szIPAddress)+1);
    pDPAddress->AddComponent(DPNA_KEY_HOSTNAME, wszIPAddress,
                            (wcslen(PlayerInfo->pwszName)+1)*sizeof(WCHAR),
                            DPNA_DATATYPE_STRING);

    // Добавление компонента порта
    pDPAddress->AddComponent(DPNA_KEY_PORT, &dwPort,
                            sizeof(DWORD), DPNA_DATATYPE_DWORD);

    // Установка информации об игроке
    ZeroMemory(&dppl, sizeof(DPN_PLAYER_INFO));
    dppl.dwSize = sizeof(DPN_PLAYER_INFO);
    dppl.dwInfoFlags = DPNINFO_NAME | DPNINFO_DATA;
    mbstowcs(wszPlayerName, szPlayerName, strlen(szPlayerName)+1);
    dppl.pwszName = wszPlayerName;
```

```

pDPClient->SetClientInfo(&dppi, NULL, NULL,
                        DPNSETCLIENTINFO_SYNC);

// Установка данных сессии
ZeroMemory(&dpad, sizeof(DPNA_APPLICATION_DESC));
dpad.dwSize = sizeof(DPNA_APPLICATION_DESC);
dpad.dwFlags = DPNSSESSION_CLIENT_SERVER;

// Установка имени сессии с преобразованием ANSI в Unicode
mbstowcs(wszSessionName, szSessionName,
          strlen(szSessionName)+1);
dpad.pwszSessionName = wszSessionName;

// Установка пароля (если надо)
if(szPassword != NULL) {
    mbstowcs(wszPassword, szPassword, strlen(szPassword)+1);
    dpad.pwszPassword = wszPassword;
    dpad.dwFlags |= DPNSSESSION_REQUIREPASSWORD;
}

```

Сейчас клиент готов установить соединение, но теперь все будет немного по-другому. Здесь используется функция **IDirectPlay8Client::Connect**, и вот прототип этой громадины:

```

HRESULT IDirectPlay8Client::Connect(
    const DPN_APPLICATION_DESC *const pdnAppDesc, // Данные сессии
    IDirectPlay8Address *const pHostAddr,         // Адрес сервера
    IDirectPlay8Address *const pDeviceInfo,        // Локальное
                                                    // устройство для
                                                    // использования
    const DPN_SECURITY_DESC *const pdnSecurity,   // NULL
    const DPN_SECURITY_CREDENTIALS *const pdnCredentials, // NULL
    const void *const pvUserConnectData,          // Данные для
                                                    // отправки при
                                                    // соединении
    const DWORD dwUserConnectDataSize,            // Размер отпра-
                                                    // ляемых данных
    void *const pvAsyncContext,                   // Контекст асин-
                                                    // хронной операции
    DPNHANDLE *const phAsyncHandle,               // Асинхронный
                                                    // дескриптор
    const DWORD dwFlags);                        // 0 или
                                                    // DPNCONNECT_SYNC

```

Я сказал, что функция **Connect** громадная, но большинство ее аргументов уже описано в предыдущем разделе «Работа с сервером». Самое явное различие — добавление аргумента **pDeviceInfo**, являющегося объектом **IDirectPlay8Address**, который содержит локальное устройство, используемое для установки соединения.

Вы узнали как получить объект **IDirectPlay8Address** в разделе «Использование адресов». Итак, остается **dwFlags**, который сообщает DirectPlay должно ли подключение работать асинхронно (0) или синхронно (**DPNCONNECT\_SYNC**).

При использовании асинхронного подключения управление возвращается немедленно, и вы должны ждать сообщения **DPN\_MSGID\_CONNECT\_COMPLETE**, сообщающего об успешном

подключении к серверу. При синхронном соединении функция возвратит управление только после установки соединения или при возникновении ошибки.

Теперь, давайте возьмем первое TCP/IP устройство в системе и передадим его этой функции, используя синхронный метод подключения:

```
IDirectPlay8Address *pDPDevice = NULL;

DPN_SERVICE_PROVIDER_INFO *pSP = NULL;
DPN_SERVICE_PROVIDER_INFO *pSPPtr;

DWORD dwSize = 0;
DWORD dwNumSP = 0;
DWORD i;

// Запрашиваем размер буфера данных
if(SUCCEEDED(pDPClient->EnumServiceProviders(
    &CLSID_DP8SP_TCPIP, NULL, pSP,
    &dwSize, &dwNumSP, 0))) {

    // Выделяем буфер и снова выполняем перечисление
    pSP = new BYTE[dwSize];
    if(SUCCEEDED(pDPClient->EnumServiceProviders(
        &CLSID_DP8SP_TCPIP, NULL, pSP,
        &dwSize, &dwNumSP, 0))) {

        // Перечисление закончено, используем
        // первый экземпляр TCP/IP
        pSPPtr = pSP;
        if(FAILED(DirectPlay8AddressCreate(
            &IID_IDirectPlay8Address,
            (void**)&pDPDevice, NULL))) {
            pDPClient->Release();
            pDPAddress->Release();
            return NULL;
        }
        pDPDevice->AddComponent(DPNA_KEY_DEVICE,
            pSPPtr->guid,
            sizeof(GUID), DPNA_DATATYPE_GUID);
    }

    // Освобождаем память буфера данных
    delete[] pSP;
}

// Устанавливаем соединение
if(FAILED(pDPClient->Connect(&dpad, &pDPAddress,
    &pDPDevice, NULL, NULL, NULL, 0, NULL, NULL,
    DPNCONNECT_SYNC))) {
    pDPAddress->Release();
    pDPDevice->Release();
    pDPClient->Release();
    return NULL;
}

// Освобождаем объект адреса - он больше не нужен
pDPAddress->Release();
pDPDevice->Release();

// Возвращаем объект клиента
return pDPClient;
}
```

## Получение и отправка сообщений

Получение сообщений на стороне клиента идентично их получению на стороне сервера, так что об этом вы заботитесь внутри функции обработчика сообщений. Что касается отправки, то это работа функции **IDirectPlay8Client::Send**:

```
HRESULT IDirectPlay8Client::Send(
    const DPN_BUFFER_DESC *const pBufferDesc, // Отправляемые данные
    const DWORD cBufferDesc,                  // 1
    const DWORD dwTimeOut,                    // Время ожидания
                                              // (в миллисекундах)
    void *const pvAsyncContext,                // Указатель на
                                              // асинхронный контекст
    DPNHANDLE *const phAsyncHandle,            // Асинхронный дескриптор
    const DWORD dwFlags);                     // Флаги (те же, что у
                                              // сервера в SendTo)
```

Вы использовали эти аргументы в разделе «Отправка сообщений сервера», так что я не буду описывать их снова. Вместо этого я приведу пример конструирования пакета игровых данных, который отправляется серверу через функцию **Send** объекта клиента.

```
#define MSG_PLAYERSTATE 0x101

typedef struct {
    DWORD dwType;
    DWORD dwXPos, dwYPos, dwHealth;
} sPlayerInfo;

sPlayerInfo PlayerData;

// Отправляем где-нибудь в программе
DPN_BUFFER_DESC dpbd;
dpbd.dwBufferSize = sizeof(sPlayerInfo);
dpbd.pBufferData = &PlayerData;
PlayerData.dwType = MSG_PLAYERSTATE;

pDPClient->Send(&dpbd, 1, 0, NULL, NULL, DPNSEND_NOCOPY);
```

## Получение идентификатора игрока

После того, как соединение с сервером установлено, наступает время, когда клиенту будет необходим доступ к его идентификатору (такую ситуацию вы увидите в главе 15, где клиент отслеживает игроков по их идентификаторам). Чтобы получить идентификатор игрока клиент должен проанализировать сообщение об *установке соединения* (**DPN\_MSGID\_CONNECT\_COMPLETE**). Это сообщение использует следующую структуру данных, содержащую важную информацию о соединении клиента и сервера:



```
typedef struct _DPNMSG_CONNECT_COMPLETE {
    DWORD dwSize; // Размер структуры
    DPNHANDLE hAsyncOp; // Дескриптор асинхронной
                        // операции
    PVOID pvUserContext; // Контекст пользователя
    HRESULT hResultCode; // Результат соединения
    PVOID pvApplicationReplyData; // Ответ на подключение
    DWORD dwApplicationReplyDataSize; // Размер данных подключения
    DPNID dpnidLocal; // Локальный идентификатор игрока
} DPNMSG_CONNECT_COMPLETE, *PDPNMSG_CONNECT_COMPLETE;
```

Сейчас вас интересует только один фрагмент информации — поле **dpnidLocal**, которое содержит локальный идентификатор игрока. Сейчас, хотя остальная информация и полезна, мы не будем иметь дела с ней. Фактически, я пропущу весь этот хлам и сосредоточусь на **dpnidLocal**.

В главе 15 вы увидите, что при установке соединения с сервером ваше клиентское приложение сохраняет значение **dpnidLocal**, чтобы использовать для ссылки на себя. Причина для ссылки на себя в том, что клиент должен хранить информацию о каждом подключенном клиенте — помните, я говорил, что идентификатор игрока является единственным реальным способом отличить одного клиента от другого.

Теперь давайте пополним багаж знаний еще одним фрагментом информации и посмотрим как завершить начатую сессию.

## Завершение сессии клиента

Когда приходит время отключить клиента от сессии, вы должны явно сообщить об этом DirectPlay, чтобы были приняты необходимые меры для разрыва соединения. Обработывается отключение клиента от сессии функцией **IDirectPlay8Client::Close**:

```
pDPClient->Close(0);
```

## Заканчиваем с сетями

Если вы еще не заметили, использовать DirectPlay очень просто, чего нельзя было ожидать от такой сложной темы, как работа с сетями, и такого мощного инструмента. Microsoft действительно закрепила интерфейс и предоставила его вам наилучшим способом.

Дальше есть только один путь. Вооружившись информацией из этой главы, вы сможете добавить в свой проект возможности, которых так жаждут игроки. Обратитесь к приложению А за ссылками на другие книги и сайты, посвященные работе с сетями — и, как всегда, просмотрите документацию DirectX SDK, где есть дополнительная информация и примеры.

## Программы на CD-ROM

Программы, демонстрирующие обсуждавшийся в этой главе код, находятся на прилагаемом к книге CD-ROM. В папке \BookCode\Chap05\ вы найдете следующие программы:

**Enum** — программа перечисляет всех поставщиков услуг и устройства системы. Местоположение: \BookCode\Chap05\Enum\.

**Server** — демонстрационное серверное приложение. Работает совместно с демонстрационным клиентом. Создает чат-сервер для обмена сообщениями между всеми подключенными клиентами. Местоположение: \BookCode\Chap05\Server\.

**Client** — демонстрационное клиентское приложение. Работает совместно с демонстрационным сервером. Подключается к серверу и начинает обмен сообщениями. Местоположение: \BookCode\Chap05\Client\.



# Глава 6

## Создаем ядро игры

Если вы читаете эту книгу с самого начала, то уже пробрались сквозь основы. Теперь вы знаете, что работа с техниками кодирования для DirectX и Windows может быть непростой задачей. При работе с этими двумя программами уловка состоит в конструировании базовой библиотеки вспомогательных функций, которые будут выполнять для вас повторяющийся от приложения к приложению код. Такая библиотека функций поможет быстро создавать игровые проекты, без необходимости писать один и тот же код для DirectX или Windows снова и снова. В этой главе я покажу вам одну такую библиотеку, которую разработал для помощи в создании демонстрационных программ к этой книге, и которую вы также можете использовать в качестве вспомогательного инструмента для ваших игровых проектов.

В главе вы узнаете о следующих вещах:

- Изучение концепций ядра.
- Создание библиотек ядра.

### Знакомство с концепцией ядра

Ядро игры — это набор библиотек, созданных мной для упрощения программирования для DirectX и Windows. В ядре игры представлены почти все функции, которые вам могут потребоваться в игровом проекте, включая функции для рисования графики, воспроизведения звуков, обработки пользовательского ввода и управления работой приложения. Это означает, что вам не надо иметь дело с низкоуровневым кодом Windows или DirectX, каждый раз когда вы начинаете новый проект!

Вместо этого, в случае необходимости, просто добавьте различные компоненты ядра к вашему игровому проекту. Вот пять модулей ядра, которые я разработал для этой книги (название каждого модуля ядра отражает его функциональность):

- **Системное ядро.** Обеспечивает взаимодействие с Windows, включая регистрацию класса окна, создание окна приложения, работу с процессами, состояниями и упаковку данных.

- **Графическое ядро.** Рисует графику как профессионал. Использует двухмерные методы для быстрой отрисовки изображений, или зажигает на сцене, используя трехмерные методы, такие как анимация сеток.
- **Ядро ввода.** Обрабатывает пользовательский ввод с клавиатуры, мыши и джойстиков.
- **Звуковое ядро.** Играет для пользователя многоканальные звуки и музыку. Меняет звук инструментов и создает уникальные партитуры.
- **Сетевое ядро.** Осуществляет подключение к Интернету и поддержку многопользовательских игр. С этим ядром вы можете присоединиться к сетевому сообществу.

---

**ПРИМЕЧАНИЕ**

Ядро игры базируется исключительно на той информации, которую вы прочитали в главах с 1 по 5. Если вам потребуется помощь по какой-либо из частей ядра, пожалуйста свободно обращайтесь к соответствующей главе за информацией о подробностях функционирования.

---

Вы можете использовать эти модули по отдельности; не требуется включать в проект их все. Если вам нужны только звуковые возможности, добавьте к проекту только звуковое ядро. Вы хотите воспользоваться преимуществами обработки состояний, предоставляемых системным ядром? Сделайте это. Выбор за вами.

Каждый модуль содержит набор классов, для которых вы можете создавать экземпляры или использовать их для наследования ваших собственных объектов. У каждого класса есть собственное уникальное назначение. Например, вы используете класс **cInputDevice** из ядра ввода для управления отдельным устройством ввода.

Прежде всего, в каждом классе ядра есть функция инициализации и функция завершения работы. Обычно они называются **Init** и **Shutdown** соответственно, но иногда у этих двух функций будут имена **Create** и **Free**. Большинство функций ядра возвращают логические значения — **TRUE** в случае успешного завершения и **FALSE** при возникновении ошибки.

В большинстве случаев перед началом использования экземпляр класса нужно инициализировать, а также вы должны вызывать функцию завершения работы экземпляра для освобождения системных ресурсов. После инициализации объект класса готов к использованию.

Ядро очень большое, так что я не могу в главе привести полный код каждого модуля. Вместо этого я предоставлю вам обзор каждого элемента объявления класса, и к тому же небольшой пример кода, показывающий как использовать каждый класс. Чтобы получить информацию об исходном коде ядра игры, обратитесь к врезке «Программы на CD-ROM» в конце этой главы; просмотр исходного кода при чтении главы может оказать неоценимую помощь.

## Системное ядро

Системное ядро обрабатывает инициализацию и поток выполнения программы обычного игрового приложения Windows, а также процессы, состояния и упаковку данных. Это то место, с которого вы начинаете разработку вашего нового игрового проекта.

## Использование объекта ядра `cApplication`

Наиболее полезный объект в системном ядре — это **`cApplication`**, который создает окно вашего приложения и управляет потоком выполнения программы. Объект регистрирует класс окна, создает окно приложения и запускает конвейер сообщений, который обрабатывает для вас посылаемые окну приложения сообщения и в случае необходимости вызывает внутренние функции класса.

При работе приложения класс **`cApplication`** вызывает три предоставляемых вами (через объявление наследуемого класса) перегруженных функции: **`Init`**, **`Shutdown`** и **`Frame`**. У каждой из функций есть свое особое назначение, с которым вы познакомитесь позже в этом разделе. Для работы с сообщениями Windows вам надо также предоставить обработчики сообщений. Об этом я тоже расскажу чуть позже.

А теперь пойдем вперед и взглянем на приведенный ниже код, содержащий объявление класса **`cApplication`**:

```
class cApplication
{
    private:
        HINSTANCE m_hInst; // Дескриптор экземпляра
        HWND m_hWnd;      // Дескриптор окна

    protected:
        char m_Class[MAX_PATH]; // Имя класса
        char m_Caption[MAX_PATH]; // Заголовок окна
        WNDCLASSEX m_wcex;      // Структура класса окна
        DWORD m_Style;          // Стилль окна
        DWORD m_XPos;           // Координата X окна
        DWORD m_YPos;           // Координата Y окна
        DWORD m_Width;          // Ширина окна по умолчанию
        DWORD m_Height;         // Высота окна по умолчанию

    public:
        cApplication();          // Конструктор

        HWND GethWnd();          // Возвращает дескриптор окна
        HINSTANCE GethInst();     // Возвращает дескриптор экземпляра

        BOOL Run();              // Исполняет код класса
        BOOL Error(BOOL Fatal, char *Text, ...); // Печать ошибок

        BOOL Move(long XPos, long YPos); // Перемещение окна
        BOOL Resize(long Width, long Height); // Изменение размера
                                           // клиентской области
        BOOL ShowMouse(BOOL Show = TRUE); // Скрывает или
                                           // отображает курсор
}
```

```
// Обработка событий по умолчанию
virtual FAR PASCAL MsgProc(HWND hWnd,      UINT uMsg,
                           WPARAM wParam, LPARAM lParam) {
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}

// Пользовательские функции, содержащие код игры
virtual BOOL Init()          { return TRUE; }
virtual BOOL Shutdown()     { return TRUE; }
virtual BOOL Frame()        { return TRUE; }
};
```

Количество вызываемых функций в классе **cApplication** минимально, поскольку он разработан фактически для того, чтобы исполнять самого себя — вы просто добавляете ваш код игры.

Чтобы использовать класс **cApplication** вы сначала должны создать класс-наследник, использующий **cApplication** в качестве базового класса. Так вы выполняете перегрузку определенных функций для соответствия вашим собственным требованиям. Перегружаемые функции, как упоминалось ранее, — это **Init**, **Shutdown** и **Frame**.

В функции **cApplication::Init** вы размещаете весь код инициализации класса, такой как загрузка данных, подготовка состояний и т.д. Противоположность **Init** — функция **cApplication::Shutdown**, которая освобождает все ранее выделенные ресурсы. Функция **Shutdown** вызывается самой последней, а функция **Init** — самой первой.

Далее следует функция **cApplication::Frame**, которая вызывается почти на каждой итерации цикла обработки сообщений (в том случае, если у Windows нет сообщений для обработки). Как можно предположить, функция **Frame** обрабатывает один кадр из вашей игры, что может включать обработку пользовательского ввода, проверку сетевых данных и рисование графики.

---

**ПРИМЕЧАНИЕ**

Каждая из упомянутых функций (**Init**, **Shutdown** и **Frame**) возвращает логическое значение, сообщаящее классу **cApplication** о том, продолжать ли выполнение программы или выйти из нее. Если какая-либо из этих функций возвращает **FALSE**, работа приложения прерывается.

---

У вас мало причин для перегрузки функций обработчиков сообщений, если вам не требуется собственный алгоритм обработки сообщений Windows. Для обработки сообщений вы перегружаете функцию **cApplication::MsgProc**, как я вскоре покажу вам.

А пока уделите минуту своего времени на работу с классом **cApplication** для быстрого создания приложения. Убедитесь, что не забыли включить в ваш проект файлы **Core\_System.h** и **Core\_System.cpp**, а также унаследовали класс вашего приложения, с которым будете работать, как показано ниже:

```
#include "Core_System.h"

class cApp : public cApplication
{
    private:
        char *m_Name;

    public:
        cApp();           // Конструктор
        BOOL Init();      // Перегруженная функция Init
        BOOL Shutdown();  // Перегруженная функция Shutdown
        BOOL Frame();     // Перегруженная функция Frame
};
```

Этот пример кода инициализирует ваше приложение регистрируя класс окна, создавая окно и запуская конвейер сообщений, в котором непрерывно вызывается функция **Frame**. Цель примера — создание буфера и сохранение в нем текста (в данном случае моего имени) и отображение этого текста в каждом кадре. Завершая работу приложения вы освобождаете текстовый буфер и выходите из программы.

```
cApp::cApp()
{
    // Присваиваем данным экземпляра
    // значения по умолчанию
    m_Name = NULL;

    // Устанавливаем стиль, местоположение,
    // ширину и высоту окна
    m_Style = WS_OVERLAPPEDWINDOW; // Стиль окна
    m_XPos = 100; // Координата X окна
    m_YPos = 20;  // Координата Y окна
    m_Width = 400; // Ширина клиентской области
    m_Height = 400; // Высота клиентской области

    // Назначаем имя класса и заголовков окна
    strcpy(m_Class, "NameClass");
    strcpy(m_Caption, "My Name Example");
}

BOOL cApp::Init()
{
    // Выделяем память для хранения имени
    if((m_Name = new char[10]) == NULL)
        strcpy(m_Name, "Jim Adams");
    else
        return FALSE;
    return TRUE;
}

BOOL cApp::Shutdown()
{
    // Освобождаем буфер имени
    delete [] m_Name;
    m_Name = NULL; // Сброс указателя
}

BOOL cApp::Frame()
{
    // Отображаем имя и ждем нажатия OK или CANCEL,
```



```
// выход из программы по CANCEL
if(MessageBox(GethWnd(), m_Name, "My name is",
             MB_OKCANCEL) == IDCANCEL)
    return FALSE;
return TRUE;
}
```

Ну вот! Все, что осталось сделать — это создать экземпляр вашего нового класса и запустить его из функции **WinMain**:

```
int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hPrev,
                  LPSTR szCmdLine, int nCmdShow)
{
    cApp App;
    return App.Run();
}
```

## Обработка состояний с **cStateManager**

В главе 1, «Подготовка к работе с книгой», было показано использование класса обработки на основе состояний. В этом разделе я применю данную информацию в качестве фундамента и покажу вам усовершенствованную версию разработанного в главе 1 класса **cStateManager**. При создании ваших игр вы обнаружите, что эта версия **cStateManager** лучше подходит для управления потоком выполнения игровой программы.

Новый диспетчер состояний добавляет пару концепций: цели вызова и добавление к функциям в **cStateManager** определяемых пользователем указателей на данные.

---

**ПРИМЕЧАНИЕ**

*Цель вызова (calling purpose)* — это, согласно названию, причина, по которой вызывается состояние. «Целью» может быть **INITPURPOSE** (сигнализирует, что функция должна подготовиться к работе), **FRAMEPURPOSE** (для обработки отдельного кадра) или **SHUTDOWNPURPOSE** (для освобождения всех ресурсов, когда обработка завершена).

---

Вот код объявления класса **cStateManager**:

```
class cStateManager
{
    // Указатели на функции состояний (связанный список)
    typedef struct sState {
        void (*Function)(void *Ptr, long Purpose);
        sState *Next;

        // Конструктор структуры, очищающий указатели
        sState() {
            Function = NULL;
            Next = NULL;
        }

        // Деструктор структуры для удаления
        // из связанного списка
        ~sState() { delete Next; Next = NULL; }
    } sState;
```

```

protected:
    sState *m_StateParent; // Родитель связанного списка
                          // стека состояний

public:
    cStateManager(); // Конструктор
    ~cStateManager(); // Деструктор

    // Помещает состояние в стек вместе с определяемым
    // пользователем указателем. Функция помещения в стек
    // вызывает функцию состояния, указывая в качестве
    // цели инициализацию
    void Push(void (*Function)(void *Ptr, long Purpose),
              void *DataPtr = NULL);

    // Выталкивание верхнего состояния из стека с вызовом
    // функции состояния для завершения работы
    BOOL Pop(void *DataPtr = NULL);

    // Выталкивание всех состояний с завершением работы
    // каждого из них
    void PopAll(void *DataPtr = NULL);

    // Обработка верхнего состояния в стеке с указанием
    // обработки отдельного кадра в качестве цели вызова
    BOOL Process(void *DataPtr = NULL);
};

```

Работа с классом **cStateManager** может сперва показаться необычной (особенно что касается целей вызова), но не следует беспокоиться, для этого и существуют примеры! Взгляните на пример, который базируется на предыдущем примере работы с **cApplication**:

```

class cApp : public cApplication
{
private:
    cStateManager m_StateManager;

    // Прототипы функций состояний
    static void Function1(void *, long);
    static void Function2(void *, long);

public:
    BOOL Init() { m_StateManager.Push(Function1, this); }
};

void cApp::Function1(void *DataPtr, long Purpose)
{
    // Получаем указатель на вызывающий класс,
    // поскольку функция статическая, а значит
    // не имеет назначенного экземпляра класса
    cApp *cc = (cApp*)DataPtr;

    // При инициализации отображаем сообщение
    // и помещаем в стек второе состояние
    if(Purpose == INITPURPOSE) {
        MessageBox(cc->GethWnd(), "State 1", "Message", MB_OK);
        cc->m_StateManager.Push(Function2, cc);
        return;
    }
}

```

```
// Принудительный выход из программы
if(Purpose == FRAMEPURPOSE)
    cc->m_StateManager.Pop(cc);
}

void cApp::Function2(void *DataPtr, long Purpose)
{
    cApp *cc = (cApp*)DataPtr;

    // Отображаем сообщение и удаляем себя из стека
    if(Purpose == FRAMEPURPOSE) {
        MessageBox(cc->GethWnd(), "State 2", "Message", MB_OK);
        cc->m_StateManager.Pop(cc);
        return;
    }
}
```

---

**ПРИМЕЧАНИЕ** Обратите внимание, что все функции состояний следуют одному и тому же прототипу функции. Убедитесь, что вы продублировали этот прототип в вашей программе:

```
void StateFunction(void *DataPtr, long Purpose);
```

---

Во время исполнения класс **cApp** помещает в стек функцию состояния (**Function1**). Сразу после помещения в стек функция состояния **Function1** будет вызвана (классом **cApp**), и в качестве цели вызова будет указана инициализация. В функции **Function1** сработает переключатель и в стек будет помещена вторая функция состояния (**Function2**). Когда функция **Function2** будет вызвана для обработки кадра, отображается сообщение, состояние удаляется из стека и выполнение программы завершается.

Обратите внимание на назначение добавленного определяемого пользователем указателя. Поскольку функции состояний в классе должны быть объявлены как статические (иначе код не скомпилируется), вы должны передать функции состояния указатель на экземпляр класса. А раз функция состояния является частью класса, вы можете свободно использовать этот указатель для доступа к данным класса (даже к закрытым данным).

## Процессы и cProcessManager

Класс **cProcessManager** во многом похож на **cStateManager**, за одним существенным исключением: для каждого кадра вызываются все находящиеся в стеке функции. Чтобы не остаться в тени ранее созданного в главе 1 класса, новый **cProcessManager** также использует цели вызова и определяемые пользователем указатели, подобно **cStateManager**.

Объявление класса **cProcessManager** идентично объявлению класса **cStateManager**. Вместо того, чтобы еще раз демонстрировать то же самое объявление, я пропущу его и перейду к примеру использования **cProcessManager**, который для каждого кадра вызывает две функции, помещенные в стек процессов:

```

class cApp : public cApplication
{
    private:
        cProcessManager m_ProcessManager;

        // Прототипы функций процессов
        static void Function1(void *, long);
        static void Function2(void *, long);

    public:
        BOOL Init() {
            m_ProcessManager.Push(Function1, this);
            m_ProcessManager.Push(Function2, this);
        }
};

void cApp::Function1(void *DataPtr, long Purpose)
{
    // Получаем указатель на вызывающий класс,
    // поскольку функция статическая, а значит
    // не имеет назначенного экземпляра класса
    cApp *cc = (cApp*)DataPtr;

    // Выводим сообщение
    if(Purpose == FRAMEPURPOSE) {
        MessageBox(cc->GethWnd(), "Process 1", "Message", MB_OK);
        return;
    }
}

void cApp::Function2(void *DataPtr, long Purpose)
{
    cApp *cc = (cApp*)DataPtr;

    // Выводим сообщение
    if(Purpose == FRAMEPURPOSE) {
        MessageBox(cc->GethWnd(), "Process 2", "Message", MB_OK);
        return;
    }
}

```

## Управление данными с cDataPackage

В главе 1 вы узнали, как использовать диспетчер упаковки данных. Теперь мы изменим созданный ранее диспетчер, добавив две функции, которые возвращают размер буфера данных и указатель на буфер данных. Вот объявление класса:

```

class cDataPackage
{
    protected:
        void *m_Buf; // Буфер данных
        unsigned long m_Size; // Размер буфера данных

    public:
        cDataPackage(); // Конструктор
        ~cDataPackage(); // Деструктор

        void *Create(unsigned long Size); // Создание буфера
        void Free(); // Освобождение буфера

```

```
    BOOL Save(char *Filename);           // Запись буфера в файл
    void *Load(char *Filename, unsigned long *Size); // Загрузка

    void *GetPtr();                       // Получение указателя
                                           // на буфер данных
    unsigned long GetSize();              // Получение размера данных
};
```

Как видите, класс **cDataPackage** остался тем же самым (чтобы посмотреть на использование класса **cDataPackage**, обратитесь к главе 1).

## Графическое ядро

Вы добрались до серьезного материала! Графическое ядро составляет значительную часть ядра игры и является самым большим и сложным объектом ядра, который мы рассмотрим. Возможности классов графического ядра представлены в таблице 6.1.

**Таблица 6.1.** Компоненты графического ядра

<i>Класс</i>	<i>Описание</i>
<b>cGraphics</b>	Обеспечивает инициализацию Direct3D, включение режимов визуализации, установку текстур, материалов и освещения.
<b>cTexture</b>	Хранит одну текстуру и предоставляет функции для рисования двухмерных фрагментов текстуры на экране.
<b>cMaterial</b>	Хранит описание одного материала.
<b>cLight</b>	Хранит описание одного источника света.
<b>cFont</b>	Служит оберткой для объекта <b>ID3DXFont</b> о котором говорилось в главе 2, «Рисование с DirectX Graphics».
<b>cVertexBuffer</b>	Упрощает работу с буферами вершин.
<b>cWorldPosition</b>	Управляет матрицей мирового преобразования, позволяя вам быстро позиционировать, масштабировать и вращать объекты.
<b>cCamera</b>	Содержит матрицу преобразования вида, которую можно изменять через интерфейс объекта.
<b>cMesh</b>	Содержит список сеток, загруженных из X-файла и их материалов. Используйте этот класс вместе с классом <b>cObject</b> .
<b>cObject</b>	Представляет один объект в трехмерном мире. Управляет ориентацией объекта, сеткой и состоянием анимации.
<b>cAnimation</b>	Содержит список анимаций, загруженных из X-файла. Используйте этот класс вместе с классом <b>cObject</b> .

Большинство классов графического ядра просты для использования, так что вы можете только скользнуть взглядом по их возможностям. Объявления классов каждого объекта ядра игры самодокументируемы, поэтому внимательно читайте их. Вы можете начать с **cGraphics** — прародителя всех объектов графического ядра.

## Графическая система и cGraphics

Вы используете **cGraphics** для установки видеорежимов, режимов визуализации, очистки устройства и многого другого. Как только объект **cGraphics** инициализирован, вы используете его совместно с почти каждым классом компонента графического ядра. Взгляните на объявление **cGraphics**:

```
class cGraphics
{
protected:
    HWND m_hWnd;           // Дескриптор родительского окна

    IDirect3D9 *m_pD3D;     // Объект Direct3D
    IDirect3DDevice9 *m_pD3DDevice; // Объект устройства

    ID3DXSprite *m_pSprite; // Объект двухмерного спрайта

    D3DDISPLAYMODE m_d3ddm; // Свойства видеорежима

    BOOL m_Windowed; // Флаг включения оконного режима
    BOOL m_ZBuffer;  // Флаг использования Z-буфера
    BOOL m_HAL;      // Флаг аппаратного ускорения

    long m_Width;    // Ширина для видеорежима
    long m_Height;   // Высота для видеорежима
    char m_BPP;      // Количество бит в пикселе

    char m_AmbientRed;   // Красная составляющая фонового света
    char m_AmbientGreen; // Зеленая составляющая фонового света
    char m_AmbientBlue;  // Синяя составляющая фонового света

public:
    cGraphics(); // Конструктор
    ~cGraphics(); // Деструктор

    // Функции для получения COM-интерфейсов
    IDirect3D9 *GetDirect3DCOM();
    IDirect3DDevice9 *GetDeviceCOM();
    ID3DXSprite *GetSpriteCOM();

    BOOL Init(); // Инициализация графического объекта
    BOOL Shutdown(); // Завершение работы графического объекта

    // Инициализация видеорежима по указанным параметрам
    BOOL SetMode(HWND hWnd, BOOL Windowed = TRUE,
                BOOL UseZBuffer = FALSE,
                long Width = 0, long Height = 0,
                char BPP = 0);

    // Функции возвращают количество видеорежимов
    // и информацию о них
```

```
long GetNumDisplayModes(D3DFORMAT Format);
BOOL GetDisplayModeInfo(long Num, D3DDISPLAYMODE *Mode),
                        D3DFORMAT Format);

// Возвращает количество бит в пикселе
// для заданного видеорежима
char GetFormatBPP(D3DFORMAT Format);

// Смотрит, существует ли указанный видеорежим.
// Задайте Format и Windowed, затем установите HAL
// в TRUE для проверки аппаратного ускорения или
// в FALSE для проверки эмуляции
BOOL CheckFormat(D3DFORMAT Format, BOOL Windowed, BOOL HAL);

BOOL Display();           // Отображает вторичный буфер
                        // (выполняя переключение страниц)

BOOL BeginScene();        // Вызывается перед началом визуализации
                        // чего-либо
BOOL EndScene();          // Вызывается, когда все визуализировано

BOOL BeginSprite();       // Вызывается чтобы разрешить рисование спрайтов
BOOL EndSprite();         // Вызывается для завершения рисования спрайтов

// Функции для очистки экрана и/или Z-буфера
BOOL Clear(long Color = 0, float ZBuffer = 1.0f);
BOOL ClearDisplay(long Color = 0);
BOOL ClearZBuffer(float ZBuffer = 1.0f);

// Функции для получения размеров экрана и количества бит в пикселе
long GetWidth();
long GetHeight();
char GetBPP();

// Функции для проверки наличия аппаратного ускорения и Z-буфера
// Используются только после установки видеорежима
BOOL GetHAL();
BOOL GetZBuffer();

// Установка нового преобразования перспективы
BOOL SetPerspective(float FOV=D3DX_PI/4,
                    float Aspect=1.3333f,
                    float Near=1.0f, float Far=10000.0f);

// Функции для установки мирового преобразования
// и преобразования вида
BOOL SetWorldPosition(cWorldPosition *WorldPos);
BOOL SetCamera(cCamera *Camera);

// Функции для установки текущих света, материала и текстуры
BOOL SetLight(long Num, cLight *Light);
BOOL SetMaterial(cMaterial *Material);
BOOL SetTexture(short Num, cTexture *Texture);

// Установка и получение компонент фонового освещения
BOOL SetAmbientLight(char Red, char Green, char Blue);
BOOL GetAmbientLight(char *Red, char *Green, char *Blue);

// Включение или выключение заданного источника света (0-n)
BOOL EnableLight(long Num, BOOL Enable = TRUE);

// Включение и выключение освещения, z-буферизации,
```

---

```
// альфа-смешивания и альфа-проверки. Для смешивания указываются
// необязательные функции смешивания
BOOL EnableLighting(BOOL Enable = TRUE);
BOOL EnableZBuffer(BOOL Enable = TRUE);
BOOL EnableAlphaBlending(BOOL Enable = TRUE,
                          DWORD Src = D3DBLEND_SRCALPHA,
                          DWORD Dest = D3DBLEND_INVSRCALPHA);
BOOL EnableAlphaTesting(BOOL Enable = TRUE);
};
```

С **cGraphics** вы можете сделать многое, но все начинается с вызова **cGraphics::Init**. Потом вы можете перечислить различные видеорежимы, или перескочить и вызвать **cGraphics::SetMode**, чтобы все закрутилось. Как минимум, функции **SetMode** требуется дескриптор родительского окна. По умолчанию будет установлен оконный режим (в противоположность полноэкранному) с выводом без Z-буфера.

---

<b>ПРИМЕЧАНИЕ</b>	Функция <b>cGraphics::SetMode</b> очень талантлива. Она определяет есть ли поддержка аппаратного ускорения и Z-буфера. Если эти возможности отсутствуют, функция <b>SetMode</b> будет использовать предоставляемую Direct3D программную эмуляцию функций трехмерной графики и отключит Z-буфер, чтобы гарантировать установку режима.
-------------------	---

---

Если вы хотите использовать полноэкранный режим, то должны присвоить параметру **Windowed** значение **FALSE** и указать допустимые значения **Width**, **Height** и количества бит в пикселе (**BPP**). Если какому-то из этих параметров вы присвоите 0, **SetMode** будет использовать текущие параметры рабочего стола. Если вы используете оконный режим, и задали значения **Width** или **Height**, то размеры родительского окна будут изменены в соответствии с новыми значениями.

Теперь вы можете недоумевать, что делать дальше. Перед тем, как что-либо визуализировать, вы должны вызвать функцию **cGraphics::BeginScene**. После того, как визуализация закончена, вызовите **cGraphics::EndScene**. Затем вызовите **cGraphics::Display** для отображения вашей графики.

Хотите очищать экран перед визуализацией? Делайте это с помощью трех функций очистки. Теперь вы можете включать источники света, материалы и текстуры (как показано в этом разделе), — **cGraphics** работает так, как я описал в главе 2, так что здесь вас ничто не должно пугать.

---

<b>ВНИМАНИЕ!</b>	Если вы не используете Z-буфер, не вызывайте функцию <b>Clear</b> , поскольку она получает значение для Z-буфера. Вместо нее используйте функцию <b>ClearDisplay</b> .
------------------	--

---

Для установки и включения освещения вы вызываете функцию **EnableLighting**. Альфа-смешивание позволяет добиться потрясающих результатов, и вы можете задавать необходимые коэффициенты смешивания



(для приемника и для источника). Альфа-проверка помогает вам рисовать эти надоедливые прозрачные текстуры (как показано в главе 2).

## Изображения и **cTexture**

Текстуры делают трехмерную графику ценящейся на вес золота. Плоские полигоны обретают жизнь с использованием полноцветных изображений. Попытка управлять списком текстур может потребовать некоторых усилий, но помощь **cTexture** сделает вашу жизнь легче:

```
class cTexture
{
protected:
    cGraphics *m_Graphics;           // Родительский cGraphics
    IDirect3DTexture9 *m_Texture;    // COM-интерфейс текстуры
    unsigned long m_Width, m_Height; // Размеры текстуры

public:
    cTexture(); // Конструктор
    ~cTexture(); // Деструктор

    IDirect3DTexture9 *GetTextureCOM(); // Возвращает COM-интерфейс
                                        // текстуры

    // Загружает текстуру из файла
    BOOL Load(cGraphics *Graphics, char *Filename,
              DWORD Transparent = 0,
              D3DFORMAT Format = D3DFMT_UNKNOWN);

    // Создает текстуру, используя указанные размеры и формат
    BOOL Create(cGraphics *Graphics,
                DWORD Width, DWORD Height, D3DFORMAT Format);

    // Конфигурирует класс cTexture на основе существующего
    // экземпляра объекта IDirect3DTexture9
    BOOL Create(cGraphics *Graphics,
                IDirect3DTexture9 *Texture);

    BOOL Free(); // Освобождает объект текстуры

    BOOL IsLoaded(); // Возвращает TRUE если текстура загружена

    long GetWidth(); // Возвращает ширину (шаг) текстуры
    long GetHeight(); // Возвращает высоту текстуры
    D3DFORMAT GetFormat(); // Возвращает формат хранения текстуры

    // Рисует двумерный фрагмент текстуры на устройстве
    BOOL Blit(long DestX, long DestY,
              long SrcX = 0, long SrcY = 0,
              long Width = 0, long Height = 0,
              float XScale = 1.0f, float YScale = 1.0f,
              D3DCOLOR Color = 0xFFFFFFFF);
};
```

Текстура в класс **cTexture** может быть загружена из двух источников: из файла с изображением на диске или из существующего объекта **IDirect3DTexture9**. Если вы загружаете изображение с диска, вызовите **cTexture::Load**. Этой функции для работы требуется пара параметров:

во-первых, ранее инициализированный объект **cGraphics**, и, во-вторых, имя загружаемого файла с изображением.

Есть еще два необязательных аргумента — цветовой ключ для прозрачности (если вы используете текстуры с прозрачными пикселями) и формат хранения. По умолчанию для **Transparent** используется значение 0, сообщаемое функции **Load**, что прозрачные пиксели не используются. Предоставление значения типа **D3DCOLOR\_RGBA** меняет ситуацию (убедитесь, что вы указали значение 255 для альфа-составляющей).

Когда используете **Format**, укажите формат хранения текстуры Direct3D, такой как **D3DFMT\_A1R5G5B5**. Помните, что если у текстуры есть прозрачные пиксели, у нее должен быть альфа-канал, так что убедитесь, что используете такой формат как **D3DFMT\_A1R5G5B5** или **D3DFMT\_A8R8G8B8**.

Наиболее вероятно, что вы будете использовать класс **cTexture** совместно с функцией **cGraphics::SetTexture** для рисования текстурированных полигонов. С другой стороны, если вы используете растровое изображение объекта текстуры для рисования непосредственно на экране, можно воспользоваться функцией **cTexture::Blit**, использующей специальный объект, называемый **ID3DXSprite**. Пока вы ничего не знаете про **ID3DXSprite** — мы познакомимся с ним в главе 7, «Использование двухмерной графики».

Сейчас я покажу, как использовать функцию **Blit**. Вам необходимо задать координаты места назначения, где текстура будет нарисована на экране, а также координаты верхнего левого угла источника, ширину, высоту, коэффициенты масштабирования и значение модуляции цвета, которое вы хотите использовать. В главе 7 мы более подробно рассмотрим использование функции **Blit**, а сейчас вот быстрый пример загрузки текстуры (называемой *texture.bmp*) и ее отображения на экране:

```
// g_Graphics = ранее инициализированный объект cGraphics
cTexture Texture;
Texture.Load(&g_Graphics, "texture.bmp");

// Рисуем текстуру в точке экрана 0,0 (используя двухмерный метод)
Texture.Blit(0,0);

Texture.Free(); // Выгружаем текстуру из памяти
```

## Цвета и cMaterial

В главе 2 обсуждалась важность использования материалов, меняющих визуальное представление визуализируемых объектов путем смены цвета рисуемых граней. Чтобы вам было проще менять цвета материалов, воспользуйтесь классом **cMaterial**:

```
class cMaterial
{
    protected:
        D3DMATERIAL9 m_Material; // Структура данных материала

    public:
        cMaterial(); // Конструктор

        D3DMATERIAL9 *GetMaterial(); // Возвращает объект D3DMATERIAL8

        // Установка и получение рассеиваемой составляющей цвета
        BOOL SetDiffuseColor(char Red, char Green, char Blue);
        BOOL GetDiffuseColor(char *Red, char *Green, char *Blue);

        // Установка и получение фоновой составляющей цвета
        BOOL SetAmbientColor(char Red, char Green, char Blue);
        BOOL GetAmbientColor(char *Red, char *Green, char *Blue);

        // Установка и получение отражаемой составляющей цвета
        BOOL SetSpecularColor(char Red, char Green, char Blue);
        BOOL GetSpecularColor(char *Red, char *Green, char *Blue);

        // Установка и получение испускаемой составляющей цвета
        BOOL SetEmissiveColor(char Red, char Green, char Blue);
        BOOL GetEmissiveColor(char *Red, char *Green, char *Blue);

        // Установка и получение мощности
        BOOL SetPower(float Power);
        float GetPower(float Power);
};
```

Как видите, класс **cMaterial** содержит одну структуру **D3DMATERIAL9** и предоставляет функции для установки и получения различных цветовых составляющих. Чтобы установить цветовую составляющую, укажите величины ее компонентов в диапазоне от 0 до 255. Для получения цветовой составляющей передайте соответствующей функции указатели на переменные типа **char**.

Вот пример использования **cMaterial** для создания желтого материала:

```
// g_Graphics = ранее инициализированный объект cGraphics
cMaterial YellowMaterial;

YellowMaterial.SetDiffuseColor(255,255,0);
YellowMaterial.SetAmbientColor(255,255,0);

g_Graphics.SetMaterial(&YellowMaterial); // Установка материала
```

---

<b>ПРИМЕЧАНИЕ</b>	При создании экземпляра класса <b>cMaterial</b> происходит очистка члена <b>m_Material</b> и создается полностью белый материал.
-------------------	--

---

Вы используете класс **cMaterial** совместно с функцией **cGraphics::SetMaterial** для установки текущего материала визуализации.

## Освещение с cLight

Источники света — простые игрушки, почти как материалы. С источниками света можно выполнять множество операций различными способами, поэтому я оборачиваю все это (по крайней мере, все о чем вы прочитали в главе 2) в класс с именем **cLight**:

```
class cLight
{
protected:
    D3DLIGHT9 m_Light; // Структура данных источника света

public:
    cLight(); // Конструктор

    D3DLIGHT9 *GetLight(); // Получение структуры данных источника света

    BOOL SetType(D3DLIGHTTYPE Type); // Установка типа источника света:
                                    // D3DLIGHT_POINT
                                    // D3DLIGHT_SPOT
                                    // D3DLIGHT_DIRECTIONAL

    // Абсолютное или относительное перемещение
    // источника света из текущей позиции
    BOOL Move(float XPos, float YPos, float ZPos);
    BOOL MoveRel(float XPos, float YPos, float ZPos);

    // Получение текущей позиции источника света
    // в указанные переменные
    BOOL GetPos(float *XPos, float *YPos, float *ZPos);

    // Установка абсолютного или относительного
    // направления лучей света
    BOOL Point(float XPos, float YPos, float ZPos);
    BOOL PointRel(float XPos, float YPos, float ZPos);

    // Получение текущего направления источника света
    // в указанные переменные
    BOOL GetDirection(float *XPos, float *YPos, float *ZPos);

    // Установка и получение различных цветовых компонент
    BOOL SetDiffuseColor(char Red, char Green, char Blue);
    BOOL GetDiffuseColor(char *Red, char *Green, char *Blue);
    BOOL SetSpecularColor(char Red, char Green, char Blue);
    BOOL GetSpecularColor(char *Red, char *Green, char *Blue);
    BOOL SetAmbientColor(char Red, char Green, char Blue);
    BOOL GetAmbientColor(char *Red, char *Green, char *Blue);

    // Установка и получение дальности освещения
    BOOL SetRange(float Range);
    float GetRange();

    // Установка и получение значения затухания
    BOOL SetFalloff(float Falloff);
    float GetFalloff();

    // Установка и получение различных коэффициентов затухания
    BOOL SetAttenuation0(float Attenuation);
    float GetAttenuation0();
    BOOL SetAttenuation1(float Attenuation);
```

```
float GetAttenuation1();
BOOL SetAttenuation2(float Attenuation);
float GetAttenuation2();

// Установка и получение значения Theta
BOOL SetTheta(float Theta);
float GetTheta();

// Установка и получение значения Phi
BOOL SetPhi(float Phi);
float GetPhi();
};
```

Чтобы использовать источники света в вашем собственном проекте вам необходимо объявить экземпляр класса **cLight**, указать тип используемого источника света (выбрав из стандартных типов источников света Direct3D, как показано в комментариях), задать цвет и местоположение (и, если необходимо, направление) где вы хотите разместить источник света. Чтобы установить источник света в сцене, воспользуйтесь функцией **cGraphics::SetLight**, показанной ранее в разделе «Графическая система и cGraphics».

---

<b>ПРИМЕЧАНИЕ</b>	Конструктор класса <b>cLight</b> создает белый точечный источник света (см. главу 2), размещенный в точке с координатами 0, 0, 0. Также учтите, что при вызове <b>Point</b> или <b>PointRel</b> вектор направления нормализуется (его компоненты будут меньше или равны 1.0, как говорилось в главе 2).
-------------------	---

---

Вот пример создания белого источника направленного света:

```
// g_Graphics = ранее инициализированный объект cGraphics
cLight DirLight;

DirLight.SetType(D3DLIGHT_DIRECTIONAL);
DirLight.Move(0.0, 10.0f, 0.0f); // Помещаем в 10 единицах над
                                // началом координат
DirLight.Point(0.0, -1.0, 0.0f); // Направляем вниз

g_Graphics.SetLight(0, &DirLight); // Устанавливаем источник света 0
g_Graphics.EnableLight(0, TRUE);   // Включаем освещение
```

## Текст и шрифты с использованием cFont

Хотя иметь дело с объектом **ID3DXFont** достаточно просто, установка шрифта может вызвать затруднения. Для быстрой и простой инициализации шрифтов в вашем проекте вы можете использовать класс **cFont**:

```
class cFont
{
private:
    ID3DXFont *m_Font; // COM-объект шрифта

public:
    cFont(); // Конструктор
```

```

~cFont(); // Деструктор

ID3DXFont *GetFontCOM(); // Возвращает COM-объект шрифта

// Создание и освобождение шрифта
BOOL Create(cGraphics *Graphics, char *Name,
            long Size = 16, BOOL Bold = FALSE,
            BOOL Italic = FALSE, BOOL Underline = FALSE,
            BOOL Strikeout = FALSE);

BOOL Free();

// Начало и завершение операций рисования текста
BOOL Begin();
BOOL End();

// Печать указанного текста
BOOL Print(char *Text, long XPos, long YPos,
            long Width = 0, long Height = 0,
            D3DCOLOR Color = 0xFFFFFFFF, DWORD Format = 0);
};

```

Чтобы начать использовать шрифт, вы должны создать его функцией **cFont::Create**. Вы передаете ей ранее инициализированный объект **cGraphics** вместе с именем шрифта (таким, как Arial или Times New Roman) и указываете высоту шрифта в пикселах. В качестве необязательных параметров вы можете указать полужирное, курсивное, подчеркнутое и перечеркнутое начертание шрифта.

Обратите внимание на пару функций **cFont::Begin** и **cFont::End**, сообщающих Direct3D, что вы собираетесь начать рисование текста, и что вы завершили печать текста, соответственно. Вам не требуется явно вызывать эти функции, поскольку метод **Print** делает это за вас (если вы не сделали это сами). Однако, отсутствие вызовов **Begin** и **End** замедляет вывод текста, поскольку функция **Print** будет постоянно вызывать **Begin** и **End** каждый раз, когда вы печатаете очередную строку текста.

Для печати строки текста вы вызываете функцию **cFont::Print**, передавая ей указатель на текст, который хотите напечатать, координаты, с которых начнется печать, размеры ограничивающего прямоугольника, по которому будет отсекается текст (по умолчанию значения **Width** и **Height** равны 0, что означает печать на всем экране), цвет текста (по умолчанию белый; используйте для задания цвета макрос **D3DCOLOR\_RGBA**) и форматирование текста (комбинация флагов, перечисленных в таблице 6.2).

**Таблица 6.2.** Флаги форматирования **cFont::Print**

Флаг	Описание
<b>DT_BOTTOM</b>	Выравнивает текст по нижнему краю ограничивающего прямоугольника.
<b>DT_CENTER</b>	Центрирует текст по горизонтали в ограничивающем прямоугольнике.

**Таблица 6.2.** Флаги форматирования cFont::Print (продолжение)

Флаг	Описание
DT_LEFT	Выравнивает текст по левому краю ограничивающего прямоугольника.
DT_NOCLIP	Рисуем текст без обрезания по ограничивающему прямоугольнику. Обеспечивает более быструю печать.
DT_RIGHT	Выравнивает текст по правому краю ограничивающего прямоугольника.
DT_TOP	Выравнивает текст по верхнему краю ограничивающего прямоугольника.
DT_WORDBREAK	Переносит слова при достижении правого края ограничивающего прямоугольника.

Вот пример создания и использования экземпляра класса **cFont**:

```
// g_Graphics = ранее инициализированный объект cGraphics
cFont ArialFont;

ArialFont.Create(&g_Graphics, "Arial");          // Шрифт Arial, размер 16
ArialFont.Print("I can print fonts!", 0, 0);     // Печать в точке 0,0
```

## Вершины и cVertexBuffer

Вершины могут быть бременем и, к сожалению, вы почти ничего не можете с этим поделать. Класс **cVertexBuffer** немного облегчает бремя, предоставляя быстрый способ создания, инициализации и визуализации наборов вершин, как показано ниже:

```
class cVertexBuffer
{
private:
    cGraphics *m_Graphics;          // Родительский cGraphics
    IDirect3DVertexBuffer9 *m_pVB; // COM-объект буфера вершин

    DWORD m_NumVertices;            // Количество вершин
    DWORD m_VertexSize;             // Размер вершины
    DWORD m_FVF;                   // Дескриптор FVF

    BOOL m_Locked;                 // Флаг блокировки буфера
    char *m_Ptr;                   // Указатель на буфер

public:
    cVertexBuffer(); // Конструктор
    ~cVertexBuffer(); // Деструктор

    // Функции для получения COM-интерфейса,
    // размера, FVF, и количества вершин
    IDirect3DVertexBuffer9 *GetVertexBufferCOM();
    unsigned long GetVertexSize();
    unsigned long GetVertexFVF();
    unsigned long GetNumVertices();
}
```

```

// Создание и освобождение буфера вершин
BOOL Create(cGraphics *Graphics,
            unsigned long NumVertices,
            DWORD Descriptor,
            long VertexSize);

BOOL Free();
BOOL IsLoaded();           // Возвращает TRUE если выделена
                           // память под буфер

// Копирует последовательность вершин в буфер
BOOL Set(unsigned long FirstVertex,
          unsigned long NumVertices, void *VertexList);

// Визуализирует буфер вершин на устройстве
BOOL Render(unsigned long FirstVertex,
             unsigned long NumPrimitives, DWORD Type);

// Блокирует и разблокирует буфер вершин для доступа
BOOL Lock(unsigned long FirstVertex = 0,
           unsigned long NumVertices = 0);
BOOL Unlock();
void *GetPtr();           // Возвращает указатель на
                           // заблокированный буфер вершин
};

```

Вы должны создать буфер вершин с помощью функции **cVertexBuffer::Create**, которой передается родительский объект **cGraphics**, количество вершин, для которых выделяется место, дескриптор настраиваемого формата вершин (FVF) и размер (в байтах) одной вершины. Да, конечно, вы все еще должны создать структуру данных вершины, чтобы работать с этим классом. Не беспокойтесь — как вы вскоре увидите, это не так трудно.

Когда вы завершите работу с экземпляром класса, освободите выделенные ему ресурсы вызовом **cVertexBuffer::Free**. Перед этим, однако, вам надо будет заполнить буфер информацией о вершинах, которую вы будете использовать, вызвав **cVertexBuffer::Set**. При вызове **cVertexBuffer::Set** используется индекс первой инициализируемой вершины, количество инициализируемых вершин и указатель на определенный вами массив структур данных вершин.

Теперь вы готовы визуализировать полигоны, используя **cVertexBuffer::Render**. Обратите внимание, что вы должны указать первую вершину, с которой начинается рисование и общее количество рисуемых примитивов (треугольных граней). Параметр **Type** используется точно также, как это делалось в главе 2 (и как описано в таблице 6.3).

**Таблица 6.3.** Флаги типов **cVertexBuffer::Render**

Флаг	Описание
<b>D3DPT_POINTLIST</b>	Набор вершин, рисуемых как пиксели.
<b>D3DPT_LINELIST</b>	Набор изолированных (несоединенных) линий.



**Таблица 6.3.** Флаги типов `cVertexBuffer::Render` (продолжение)

Флаг	Описание
<code>D3DPT_LINESTRIP</code>	Последовательность соединенных линий. Каждая линия рисуется от предыдущей вершины к текущей, подобно игре «соедини точки».
<code>D3DPT_TRIANGLELIST</code>	Набор треугольников по три вершины на треугольник.
<code>D3DPT_TRIANGLESTRIP</code>	Полоса треугольников. Для создания грани используется одна новая вершина и две из предыдущего треугольника.
<code>D3DPT_TRIANGLEFAN</code>	Веер треугольников. Каждая вершина для создания грани соединяется с центральной вершиной и с предыдущей вершиной.

Предположим, вы хотите создать простой буфер вершин, в котором будут использоваться преобразованные двухмерные вершины (вершины, описанные в экранных координатах) с рассеиваемой цветовой составляющей. Чтобы сделать пример проще, создадим квадрат, состоящий из двух треугольников (воспользуемся полосой треугольников):

```
// g_Graphics = ранее инициализированный объект cGraphics
// Определяем структуру данных вершины и дескриптор FVF
typedef struct sVertex {
    float x, y, z, rhw;
    D3DCOLOR Diffuse;
} sVertex;
#define VERTEXFVF (D3DFVF_XYZRHW | D3DFVF_DIFFUSE)

cVertexBuffer g_VB;
g_VB.Create(&g_Graphics, 4, VERTEXFVF, sizeof(sVertex));

// Вершины полосы треугольников в порядке по часовой стрелке
sVertex Verts[4] = {
    { 0.0f, 0.0f, 0.0f, 1.0f, D3DCOLOR_RGBA(255, 0, 0, 255) },
    { 200.0f, 0.0f, 0.0f, 1.0f, D3DCOLOR_RGBA( 0, 255, 0, 255) },
    { 0.0f, 200.0f, 0.0f, 1.0f, D3DCOLOR_RGBA( 0, 0, 255, 255) },
    { 200.0f, 200.0f, 0.0f, 1.0f, D3DCOLOR_RGBA(255, 255, 255, 255) },
};
g_VB.Set(0, 4, (void*)&Verts);

// Визуализируем полосу треугольников
g_VB.Render(0, 2, D3DPT_TRIANGLESTRIP);

// Освобождаем буфер вершин
g_VB.Free();
```

## Мировое преобразование с `cWorldPosition`

Хотя работать с матрицей мирового преобразования не сложно, разве не замечательно было бы иметь класс, заботящийся обо всех деталях — таких, как мировые координаты, значения вращения и коэффициенты масштабирования? Как насчет того, чтобы добавить вращение щитов, только для аромата?

**И ВОТ ВСЕ ЭТО В cWorldPosition:**

```

class cWorldPosition
{
protected:
    BOOL m_Billboard; // Флаг использования щита

    // Текущие местоположение, поворот и масштаб
    float m_XPos, m_YPos, m_ZPos;
    float m_XRotation, m_YRotation, m_ZRotation;
    float m_XScale, m_YScale, m_ZScale;

    D3DXMATRIX m_matWorld;        // Матрица мирового преобразования
    D3DXMATRIX m_matScale;        // Матрица масштабирования
    D3DXMATRIX m_matRotation;     // Матрица вращения
    D3DXMATRIX m_matTranslation;  // Матрица перемещения

    D3DXMATRIX *m_matCombine1;    // Комбинационная матрица 1
    D3DXMATRIX *m_matCombine2;    // Комбинационная матрица 2

public:
    cWorldPosition(); // Конструктор

    // Возвращает матрицу мирового преобразования
    D3DXMATRIX *GetMatrix(cGraphics *Graphics = NULL);

    // Установка внешних матриц для комбинирования с мировой матрицей
    BOOL SetCombineMatrix1(D3DXMATRIX *Matrix = NULL);
    BOOL SetCombineMatrix2(D3DXMATRIX *Matrix = NULL);

    BOOL Copy(cWorldPosition *DestPos); // Копирование в другой класс

    // Перемещение в мировых координатах (и относительно текущих)
    BOOL Move(float XPos, float YPos, float ZPos);
    BOOL MoveRel(float XAdd, float YAdd, float ZAdd);

    // Установка значений поворота (и относительно текущих)
    BOOL Rotate(float XRot, float YRot, float ZRot);
    BOOL RotateRel(float XAdd, float YAdd, float ZAdd);

    // Установка коэффициентов масштабирования (и относительно текущих)
    BOOL Scale(float XScale, float YScale, float ZScale);
    BOOL ScaleRel(float XAdd, float YAdd, float ZAdd);

    // Обновление матриц и предоставление объекта cGraphics для щита
    BOOL Update(cGraphics *Graphics = NULL);

    // Разрешение и запрещение использования щитов
    BOOL EnableBillboard(BOOL Enable = TRUE);

    // Получение текущей позиции, поворота и масштаба
    float GetXPos();
    float GetYPos();
    float GetZPos();
    float GetXRotation();
    float GetYRotation();
    float GetZRotation();
    float GetXScale();
    float GetYScale();
    float GetZScale();
};

```

Большинство функций самодокументируемые; вопросы вызывают только **Update**, **GetMatrix**, **SetCombineMatrix1** и **SetCombineMatrix2**. Функция **Update** заново строит матрицу мирового преобразования, используя хранящуюся ориентацию и учитывая матрицу щита и две внешние матрицы (которые я называю *комбинационными матрицами*). Для установки источников двух комбинационных матриц используются функции **SetCombineMatrix1** и **SetCombineMatrix2**.

---

<b>ПРИМЕЧАНИЕ</b>	Комбинационные матрицы (или матрицы привязки) представляют преобразования, необходимые для присоединения одного объекта к другому, например, для присоединения оружия к сетке руки. Две матрицы представляют локальную ориентацию фрейма присоединяемой сетки и мировую ориентацию присоединяемой сетки соответственно.
-------------------	---

---

Функция **GetMatrix** возвращает текущую матрицу мирового преобразования. Убедитесь, что передали функции **GetMatrix** текущий объект **cGraphics**, который используется для вычисления матрицы щита (она вычисляется из транспонированной матрицы вида).

Вот пример ориентации двух объектов в трехмерном мире (один присоединяется к другому):

```
cWorldPosition ObjectPos, ObjectPos2;

ObjectPos.Move(10.0f, 100.0f, -56.0f);
ObjectPos.Rotate(1.57f, 0.0f, 0.785f);
ObjectPos.Update(); // Вычисляем обновленную матрицу

// Комбинируем второй объект с первым (наследование ориентации)
ObjectPos2.SetCombineMatrix1(ObjectPos.GetMatrix());
ObjectPos2.Rotate(0.0f, 0.0f, 3.14f);
ObjectPos2.Update(); // Вычисляем обновленную матрицу,
                    // используя комбинирование
```

## Преобразование вида и cCamera

Класс **cCamera**, во многом похожий на **cWorldPosition**, имеет дело с матрицей преобразования вида:

```
class cCamera
{
    protected:
        float m_XPos, m_YPos, m_ZPos; // Координаты местоположения
        float m_XRot, m_YRot, m_ZRot; // Значения поворота

        // Отслеживание ориентации камеры
        float m_StartXPos, m_StartYPos, m_StartZPos;
        float m_StartXRot, m_StartYRot, m_StartZRot;
        float m_EndXPos, m_EndYPos, m_EndZPos;
        float m_EndXRot, m_EndYRot, m_EndZRot;
```

```

D3DXMATRIX m_matWorld;          // Матрица мирового преобразования
D3DXMATRIX m_matTranslation;    // Матрица перемещения
D3DXMATRIX m_matRotation;       // Матрица вращения

public:
    cCamera(); // Конструктор

    D3DXMATRIX *GetMatrix(); // Получение матрицы преобразования вида
    BOOL Update();           // Обновление матрицы преобразования

    // Перемещение и вращение камеры (вида)
    BOOL Move(float XPos, float YPos, float ZPos);
    BOOL MoveRel(float XAdd, float YAdd, float ZAdd);
    BOOL Rotate(float XRot, float YRot, float ZRot);
    BOOL RotateRel(float XAdd, float YAdd, float ZAdd);

    // Направляем камеру из точки Eye на точку At
    BOOL Point(float XEye, float YEye, float ZEye,
               float XAt, float YAt, float ZAt);

    // Установка начала и конца отслеживания ориентации
    BOOL SetStartTrack();
    BOOL SetEndTrack();

    // Интерполяция ориентации камеры вдоль траектории
    // с использованием времени (0.0 – 1.0) и общей длины
    BOOL Track(float Time, float Length);

    // Получение значений перемещения и вращения
    float GetXPos();
    float GetYPos();
    float GetZPos();
    float GetXRotation();
    float GetYRotation();
    float GetZRotation();
};

```

Класс **cCamera** работает во многом также как класс **cWorldPosition**, так что я воздержусь от вводных слов. Единственное отличие заключается в добавлении функций **Point**, **SetStartTrack**, **SetEndTrack** и **Track**. Функцию **Point** вы используете для ориентирования порта просмотра и его мгновенного направления на заданную точку.

Трио относящихся к траектории камеры функций отслеживает перемещение камеры с течением времени. Чтобы использовать возможности отслеживания ориентации камеры поместите камеру в начальное положение и вызовите функцию **cCamera::SetStartTrack**. Затем переместите камеру в конечную позицию и вызовите **cCamera::SetEndTrack**.

Теперь понятна причина вызова **cCamera::Track** (перед вызовом **cCamera::Update**) — ориентирование камеры согласно созданной вами траектории. Параметр **Time** функции **Track** меняется от 0.0 (начальная ориентация) до 1.0 (конечная ориентация) и любое значение между этими двумя перемещает камеру вдоль траектории. Параметр **Length** может быть любым значением, с которым вы работаете (например, миллисекундами).

Отслеживание камеры может создать сногшибательные эффекты, так что давайте перейдем к примеру:

```
cCamera Cam;

// Размещаем камеру в точке 0.0f, 100.0f, -100.0f
// и направляем на начало координат
Cam.Point(0.0f, 100.0f, -100.0f, 0.0f, 0.0f, 0.0f);
Cam.SetStartTrack();

// Перемещаем в конечную позицию
Cam.Point(-100.0f, 0.0f, 0.0f, 0.0f, 100.0f, 0.0f);
Cam.SetEndTrack();

// Помещаем камеру на полпути через 10000 миллисекунд
Cam.Track(0.5f, 10000);
Cam.Update();
```

Чтобы установить текущую матрицу преобразования вида согласно местоположению камеры, используйте функцию **cGraphics::SetCamera**:

```
g_Graphics.SetCamera(&Cam); // Не вызывайте преждевременно Update
```

## Загружаемые сетки и использование cMesh

Теперь вы не скажете, что работать с сетками не трудно, правда? Конечно, я говорю о скелетных и обычных сетках, которые вы использовали в главе 2. Цель **cMesh** — заключить этих маленьких демонов в набор простых для использования классов и затем применять их с другими объектами, которые визуализируют сетки на экране.

```
class cMesh
{
private:
    cGraphics *m_Graphics; // Родительский объект cGraphics

    long m_NumMeshes; // Количество сеток в классе
    sMesh *m_Meshes; // Список сеток

    long m_NumFrames; // Количество фреймов в классе
    sFrame *m_Frames; // Список фреймов

    D3DXVECTOR3 m_Min, m_Max; // Координаты ограничивающего
                                // параллелепипеда
    float m_Radius; // Радиус ограничивающей сферы

    // Функция выполняет разбор отдельного шаблона X-файла
    void ParseXFileData(IDirectXFileData *pData,
                        sFrame *ParentFrame, char *TexturePath);

    // Сопоставление костей и матриц преобразования фреймов
    void MapFramesToBones(sFrame *Frame);

public:
    cMesh(); // Конструктор
    ~cMesh(); // Деструктор
```

```

    BOOL IsLoaded();          // Возвращает TRUE если сетка загружена

    long   GetNumFrames();     // Возвращает количество фреймов
                                // в списке
    sFrame *GetParentFrame();  // Возвращает самый верхний фрейм
                                // из списка
    sFrame *GetFrame(char *Name); // Поиск фрейма в списке
    long   GetNumMeshes();     // Возвращает количество сеток
                                // в списке
    sMesh  *GetParentMesh();   // Возвращает самую верхнюю сетку
                                // из списка
    sMesh  *GetMesh(char *Name); // Поиск сетки в списке

    // Получаем координаты ограничивающего параллелепипеда
    // и радиус ограничивающей сферы
    BOOL GetBounds(float *MinX, float *MinY, float *MinZ,
                   float *MaxX, float *MaxY, float *MaxZ,
                   float *Radius);

    // Загрузка и освобождение .X-файла
    // (можно задать необязательный путь к карте текстур)
    BOOL Load(cGraphics *Graphics, char *Filename,
              char *TexturePath = ".\\");
    BOOL Free();
};

```

Класс выглядит небольшим, но я не показал вам структуры **sMesh** и **sFrame**, которые использует класс **cMesh**. Эти две структуры являются связанными списками объектов сеток и определений фреймов. Они также хранят различные ориентации для фреймов и списки материалов и текстур. Пойдемте дальше и загрузим исходный код графического ядра, чтобы взглянуть на него; он хорошо прокомментирован и вам будет нетрудно в нем разобраться.

Единственная вещь, которую вы будете делать с **cMesh**, это использование его для загрузки сеток из X-файлов, как показано ниже:

```

// g_Graphics = ранее инициализированный объект cGraphics
cMesh Mesh;

Mesh.Load(&g_Graphics, "Mesh.x");

Mesh.Free(); // Освобождаем сетку, когда закончили с ней работать

```

## Рисуем объекты, используя cObject

Когда приходит время рисовать сетки, вы должны создать мост от определения сетки до экрана. «Почему нельзя поддерживать визуализацию, используя объект cMesh», спрашиваете вы? Ответ — расходование памяти. Что если вы хотите использовать одну и ту же сетку снова и снова? Решение — применить объект **cObject**:

```

class cObject
{
    protected:
        cGraphics *m_Graphics; // Родительский объект cGraphics

```

```
cMesh *m_Mesh; // Сетка для рисования
sAnimationSet *m_AnimationSet; // Анимационный набор
cWorldPosition m_Pos; // Мировая ориентация

BOOL m_Billboard; // Флаг объекта щита

unsigned long m_StartTime; // Начальное время анимации

// Функции обновляют ориентацию фреймов и рисуют сетки
void UpdateFrame(sFrame *Frame, D3DXMATRIX *Matrix);
void DrawFrame(sFrame *Frame);
void DrawMesh(sMesh *Mesh);

public:
    cObject(); // Конструктор
    ~cObject(); // Деструктор

    // Создаем и освобождаем объект
    // (с необязательным указанием сетки)
    BOOL Create(cGraphics *Graphics, cMesh *Mesh = NULL);
    BOOL Free();

    // Включаем и выключаем поддержку для щитов
    BOOL EnableBillboard(BOOL Enable = TRUE);

    // Присоединяем объект к фрейму другого объекта
    // (Это вызовет комбинирование матриц при обновлении)
    // По умолчанию к первому фрейму найденному в сетке.
    BOOL AttachToObject(cObject *Object,
        char *FrameName = NULL);

    // Ориентация объекта
    BOOL Move(float XPos, float YPos, float ZPos);
    BOOL MoveRel(float XAdd, float YAdd, float ZAdd);
    BOOL Rotate(float XRot, float YRot, float ZRot);
    BOOL RotateRel(float XAdd, float YAdd, float ZAdd);
    BOOL Scale(float XScale, float YScale, float ZScale);
    BOOL ScaleRel(float XAdd, float YAdd, float ZAdd);

    D3DXMATRIX *GetMatrix(); // Получение матрицы объекта

    // Получение ориентации объекта
    float GetXPos();
    float GetYPos();
    float GetZPos();
    float GetXRotation();
    float GetYRotation();
    float GetZRotation();
    float GetXScale();
    float GetYScale();
    float GetZScale();

    // Получение масштабированного ограничивающего параллелепипеда
    // и радиуса ограничивающей сферы
    BOOL GetBounds(float *MinX, float *MinY, float *MinZ,
        float *MaxX, float *MaxY, float *MaxZ,
        float *Radius);

    // Установка сетки, которую будет рисовать класс
    BOOL SetMesh(cMesh *Mesh);

    // Установка новой анимации (с именем и начальным временем)
```

```

    BOOL SetAnimation(cAnimation *Animation,
                     char *Name = NULL,
                     unsigned long StartTime = 0);
    char *GetAnimation(); // Получение указателя на имя анимации

    // Сброс воспроизведения анимации и установка
    // нового времени начала
    BOOL ResetAnimation(unsigned long StartTime = 0);

    // Обновление анимации на основе времени
    // с использованием плавной интерполяции
    BOOL UpdateAnimation(unsigned long Time, BOOL Smooth = TRUE);

    // Возвращает TRUE если анимация завершена во время Time
    BOOL AnimationComplete(unsigned long Time);

    BOOL Update(); // Обновление матрицы преобразования объекта
    BOOL Render(); // Рисуем объект, используя мировое преобразование
};

```

Класс **cObject** предоставляет почти все, что нужно для работы с трехмерными объектами в вашем мире. Вы можете ориентировать объекты, устанавливать новые сетки, выравнивать объекты относительно зрителя (использовать щиты), устанавливать и обновлять анимации, присоединять одни объекты к другим и получать координаты ограничивающего параллелепипеда и радиус ограничивающей сферы.

#### ПРИМЕЧАНИЕ

*Ограничивающий параллелепипед (bounding box)* — это набор координат, представляющих максимальные значения координат вершин сетки. Например, если у самой высокой вершины сетки значение координаты Y равно 100.0, то именно это значение будет использовано для верхней поверхности ограничивающего параллелепипеда. То же самое верно для левой, правой, нижней, передней и задней граней ограничивающего параллелепипеда сетки. *Ограничивающая сфера (bounding sphere)* — это почти то же самое, но вместо того, чтобы использовать параллелепипед, мы используем сферу, заключающую в себе сетку.

Ограничивающие параллелепипеды и сферы очень полезны при обнаружении столкновений, то есть когда надо узнать, столкнулся ли один объект с другим.

Чтобы работать с **cObject** просто объявите экземпляр класса и присоедините к нему ранее загруженный объект сетки. Затем вы можете произвольным образом ориентировать объект и визуализировать его на экране, как показано в следующем примере:

```

// g_Graphics = ранее инициализированный объект cGraphics
// g_Mesh = ранее загруженный объект сетки
cObject g_Object;

g_Object.Create(&g_Graphics, &g_Mesh);
g_Object.Move(100.0f, 50.0f, 100.0f);
g_Object.Render();

```



## Делаем сетки двигающимися с cAnimation

Завершает описание графического ядра **cAnimation** — компонент для анимации сеток. Благодаря **cAnimation** вы можете загрузить комплект анимационных последовательностей из X-файла и использовать их совместно с **cObject** для анимации сеток.

Класс **cAnimation** небольшой. Подобно **cMesh**, **cAnimation** содержит несколько структур, хранящих список данных анимации. Взгляните на объявление класса:

```
class cAnimation
{
protected:
    long          m_NumAnimations; // Количество анимаций в классе
    sAnimationSet *m_AnimationSet; // Список анимаций

    // Разбираем отдельный шаблон X-файла
    void ParseXFileData(IDirectXFileData *DataObj,
                        sAnimationSet *ParentAnim,
                        sAnimation *CurrentAnim);

public:
    cAnimation(); // Конструктор
    ~cAnimation(); // Деструктор

    BOOL IsLoaded(); // Возвращает TRUE, если анимация загружена

    // Возвращает количество анимаций в списке,
    // верхнюю анимацию и длину анимации
    long GetNumAnimations();
    sAnimationSet *GetAnimationSet(char *Name = NULL);
    unsigned long GetLength(char *Name = NULL);

    // Загрузка и освобождение анимации (можно указать сетку,
    // к которой будет применена анимация)
    BOOL Load(char *Filename, cMesh *MapMesh = NULL);
    BOOL Free();

    BOOL MapToMesh(cMesh *Mesh); // Применение анимации к сетке

    // Включение и выключение заикленной анимации
    BOOL SetLoop(BOOL ToLoop, char *Name = NULL);
};
```

В классе **cAnimation** можно использовать только четыре общих функции: **Load**, **Free**, **MapToMesh** и **SetLoop**. Привязка анимации к сетке необходима, чтобы класс анимации мог найти матрицы сетки, которые ему следует изменять. Что касается **SetLoop**, обратите внимание на параметр **Name**, задающий имя анимации для которой будет установлен циклический повтор.

Анимациям (также как фреймам и сеткам) внутри X-файла могут быть присвоены имена. Это позволяет запаковать в один X-файл несколько анимаций и обращаться к ним по имени. Например, если ваш X-файл содержит анимационный набор с именем **Walk**, вы можете передать строку

"Walk" в параметре **Name**. Если значение **Name** равно **NULL**, будет выбрана самая верхняя анимация в списке.

Другая вещь, которую вы обязаны заметить, — параметр **StartTime** в классе **cObject**. Этот параметр задает исходное значение, которое анимация использует при определении времени анимации. Таким способом, если вы синхронизируете ваши анимации по времени (используя такие функции, как **timeGetTime**), вы можете установить **StartTime** на то время, когда должно начаться воспроизведение анимации.

#### ПРИМЕЧАНИЕ

При воспроизведении анимации время произвольно; вы можете измерять время в секундах, миллисекундах, кадрах и т.д. Вы сами должны принять решение, поддерживать и измерять время согласно вашим потребностям.

Тогда последующие вызовы **cObject::UpdateAnimation** будут использовать разницу между указанным вами в их вызове временем и значением **StartTime**, предоставляя вам ясный механизм синхронизации (другими словами, точную синхронизацию, у которой время начала воспроизведения равно 0, а не произвольному значению).

И, наконец, последний пример использования графического ядра, который загружает анимацию, зацикливает ее и применяет объект анимации к ранее созданному трехмерному объекту:

```
// g_Graphics = ранее инициализированный объект cGraphics
// g_Mesh      = загруженный объект cMesh
// g_Object    = загруженный объект cObject
cAnimation Anim;

// Загрузка и зацикливание анимации ходьбы
Anim.Load("Mesh.x", &g_Mesh);
Anim.SetLoop(TRUE, "Walk");

// Применение анимации ходьбы к объекту
g_Object.SetAnimation(&Anim, "Walk", timeGetTime());

// Вход в цикл визуализации объекта и
// обновления анимации в каждом кадре
g_Object.UpdateAnimation(timeGetTime(), TRUE);
g_Object.Render();

// Завершив работу освобождаем анимацию
Anim.Free();
```

## Ядро ввода

Уф! Графическое ядро обширно, и может понадобится время, чтобы полностью разобраться с ним. А пока давайте отвлечемся и посмотрим на ядро ввода, которое вы будете использовать, чтобы предоставить игрокам возможность управлять программой через клавиатуру, мышь и джойстик.

Устройства ввода представляются двумя простыми классами: **cInput** и **cInputDevice**. Класс **cInput** вы используете для инициализации DirectInput, а класс **cInputDevice** содержит единственный объект интерфейса устройства DirectInput. Если у вас несколько устройств, используйте отдельные объекты **cInputDevice** для каждого из них.

## Использование DirectInput с cInput

Первый этап использования системы ввода — инициализация DirectInput, являющаяся назначением класса **cInput**. Класс исключительно компактный, и его объявление выглядит так:

```
class cInput
{
protected:
    HWND      m_hWnd; // Дескриптор окна-владельца
    IDirectInput8 *m_pDI; // Интерфейс DirectInput

public:
    cInput(); // Конструктор
    ~cInput(); // Деструктор

    IDirectInput8 *GetDirectInputCOM(); // Возвращает COM-объект DI
    HWND GethWnd(); // Возвращает дескриптор окна
    BOOL Init(HWND hWnd, HINSTANCE hInst); // Инициализация класса
    BOOL Shutdown(); // Освобождение класса
};
```

Класс **cInput** исключительно легкий, и вы будете вызывать в нем только две функции (**Init** и **Shutdown**). Настоящее волшебство начинается когда вы используете класс **cInputDevice**.

## Устройства ввода и cInputDevice

Класс **cInputDevice** — вот где происходит настоящая работа. Класс **cInputDevice** используется для инициализации указанного устройства ввода (клавиатуры, мыши или джойстика) и предоставляет средства получения данных от этого устройства для использования в вашей игре. Класс **cInput** был простой, а класс **cInputDevice** сосредотачивает всю остальную функциональность системы ввода, и его объявление выглядит так:

```
class cInputDevice
{
public:
    cInput *m_Input; // Родительский класс cInput
    short m_Type; // Тип устройства
                  // MOUSE, KEYBOARD,
                  // или JOYSTICK

    IDirectInputDevice8 *m_pDIDevice; // COM-объект устройства
    BOOL m_Windowed; // TRUE, если используются методы чтения
                    // состояния мыши из Windows или FALSE,
                    // если используются методы DirectInput
};
```

```

char m_State[256]; // Состояния всех клавиш
                  // и кнопок

DIMOUSESTATE *m_MouseState; // Состояние мыши
DIJOYSTATE *m_JoystickState; // Состояние джойстика
BOOL m_Locks[256]; // Флаги блокировки
                  // клавиш или кнопок

// Координаты мыши/джойстика
long m_XPos, m_YPos;

// Внутренняя функция перечисления
static BOOL FAR PASCAL EnumJoysticks(
    LPCDIDEVICEINSTANCE pdInst, LPVOID pvRef);

public:
    cInputDevice(); // Конструктор
    ~cInputDevice(); // Деструктор

    IDirectInputDevice8 *DeviceCOM(); // Возвращает COM-объект

    // Функции для создания интерфейса устройства и его освобождения
    BOOL Create(cInput *Input, short Type,
        BOOL Windowed = TRUE);
    BOOL Free();

    BOOL Clear(); // Очистка данных устройства
    BOOL Read(); // Чтение данных устройства
    BOOL Acquire(BOOL Active = TRUE); // Захват или освобождение
    // устройства

    BOOL GetLock(char Num); // Получение состояния
    // блокировки клавиши/кнопки
    BOOL SetLock(char Num, BOOL State = TRUE); // Установка блокировки

    long GetXPos(); // Получаем позицию по x мыши/джойстика
    BOOL SetXPos(long XPos); // Устанавливаем позицию по x
    long GetYPos(); // Получаем позицию по y мыши/джойстика
    BOOL SetYPos(long YPos); // Устанавливаем позицию по y
    long GetXDelta(); // Получаем изменение по x
    // (относительное перемещение)
    long GetYDelta(); // Получаем изменение по y
    // (относительное перемещение)

    // Функции, специфичные для клавиатуры
    BOOL GetKeyState(char Num); // Получение состояния клавиши.
    // Возвращает:
    // TRUE=Нажата или FALSE=Отпущена
    // Используйте Num = KEY_* или DIK_*
    BOOL SetKeyState(char Num, BOOL State); // Установка состояния
    // клавиши
    BOOL GetPureKeyState(char Num); // Получение состояния
    // клавиши без учета
    // блокировки
    short GetKeyPress(long TimeOut = 0); // Ждем нажатия клавиши
    // и возвращаем ASCII-код
    long GetNumKeyPresses(); // Получение количества
    // нажатых в данный момент
    // клавиш
    long GetNumPureKeyPresses(); // Получение количества
    // нажатых клавиш без учета
    // блокировки

```

```
// Функции, специфичные для мыши/джойстика
BOOL GetButtonState(char Num); // Получение состояния кнопок
                                // Num = LBUTTON, RBUTTON, MBUTTON
BOOL SetButtonState(char Num, BOOL State); // Установка состояния
BOOL GetPureButtonState(char Num); // Получение состояния
                                    // без учета блокировок
long GetNumButtonPresses(); // Получение количества
                              // нажатых кнопок
long GetNumPureButtonPresses(); // Получение количества
                                 // нажатых кнопок без учета
                                 // блокировок
};
```

В классе **cInputDevice** есть все! Он охватывает все устройства — клавиатуру, мышь и джойстик — в одном удобном пакете. Объект класса начинает работу с вызова **cInputDevice::Create**, которому передается ранее инициализированный объект класса **cInput**. Вам надо также сообщить классу с каким устройством вы намереваетесь работать, указав соответствующее значение (**KEYBOARD**, **MOUSE** или **JOYSTICK**) в переменной **Type**. И, наконец, вам надо уведомить класс желаете ли вы использовать функции чтения состояния устройств из **DirectInput** или предпочтете методы **Windows**.

Установка флага **Windowed** в **TRUE** заставляет объект класса использовать функции чтения состояния устройств из **Windows**, в то время, как значение **FALSE** заставляет использовать **DirectInput**. Если вы планируете использовать оконный режим (или хотите, чтобы курсор **Windows** был виден), убедитесь, что для **Windowed** задали значение **TRUE**.

Перейдем к списку функций класса: вы вызываете **cInputDevice::Read**, чтобы прочесть текущее состояние устройства. Затем вы можете проверить состояние каждой отдельной клавиши или кнопки, используя функции **cInputDevice::GetKeyState**, **cInputDevice::GetButtonState**, **cInputDevice::GetPureKeyState** и **cInputDevice::GetPureButtonState**.

Причина наличия двух наборов функций в том, что клавиши и кнопки могут быть заблокированы. Заблокированная клавиша или кнопка не срабатывает до тех пор, пока не будет отпущена. Чтение чистых значений игнорирует состояние блокировки.

Вызов **GetKeyState** или **GetButtonState** возвращает **TRUE**, если клавиша нажата и **FALSE** — если нет. Параметр **Num** в этих функциях проверки состояния представляет проверяемую клавишу или кнопку. На клавиши ссылаются по имени с префиксом **KEY\_**, например, **KEY\_ESC** или **KEY\_A**. Полный список значений **KEY\_\*** находится в файле **Core\_Input.h** (или используйте такие константы, как **DIK\_A** и **DIK\_ESCAPE**, предоставляемые **DirectInput**).

Для ссылки на кнопки мыши используйте значения **MOUSE\_LBUTTON** (левая кнопка), **MOUSE\_RBUTTON** (правая кнопка) и **MOUSE\_MBUTTON**

(средняя кнопка). Для джойстика используйте **JOYSTICK\_BUTTON0**, **JOYSTICK\_BUTTON1**, **JOYSTICK\_BUTTON2**, **JOYSTICK\_BUTTON3**, **JOYSTICK\_BUTTON4** и **JOYSTICK\_BUTTON5**.

## Использование ядра ввода

Использовать ядро ввода просто; достаточно создать экземпляр класса **cInput** и столько объектов **cInputDevice**, сколько вам нужно, убедившись, что не забыли инициализировать каждый из них. Предположим, вы хотите использовать два устройства — клавиатуру и мышь:

```
// Глобальные объявления
cInput      g_Input;
cInputDevice g_Keyboard;
cInputDevice g_Mouse;

// Инициализация системы ввода (требуется)
// Предполагается, что hWnd и hInst уже инициализированы
// hWnd = дескриптор окна, hInst = дескриптор экземпляра
g_Input.Init(hWnd, hInst);

// Создаем устройства клавиатуры и мыши
// Для чтения мыши используем методы DirectInput
g_Keyboard.Create(&g_Input, KEYBOARD);
g_Mouse.Create(&g_Input, MOUSE, FALSE);

// Читаем текущее состояние устройств
g_Keyboard.Read();
g_Mouse.Read();

// Если нажата ESC выводим сообщение
if(g_Keyboard.GetKeyState(KEY_ESC) == TRUE) {
    // Блокируем клавишу ESC чтобы пользователь должен был
    // отпустить ее, прежде чем мы снова получим данные о нажатии
    g_Keyboard.SetLock(KEY_ESC, TRUE);
    MessageBox(hWnd, "ESCAPE", "Key Pressed!", MB_OK);
}

// Если нажата левая кнопка мыши, отображаем координаты
if(g_Mouse.GetPureButtonState(MOUSE_LBUTTON) == TRUE) {
    char b[200];
    sprintf(b, "%ld, %ld", g_Mouse.GetXPos(), g_Mouse.GetYPos());
    MessageBox(hWnd, b, "Mouse Coordinates", MB_OK);
}

// Освобождаем все
g_Mouse.Free();
g_Keyboard.Free();
g_Input.Shutdown();
```

## Звуковое ядро

Что за игра без музыки и звуков? Звуковое ядро — это решение для быстрого и легкого добавления в вашу игру звуков и музыки. В звуковое ядро входят шесть классов компонентов (см. таблицу 6.4).

Таблица 6.4. Классы звукового ядра

Класс	Описание
<b>cSound</b>	Содержит объекты DirectSound и DirectMusic и управляет звуковыми потоками.
<b>cSoundData</b>	Класс хранит волновые данные, используемые для воспроизведения с <b>cSoundChannel</b> .
<b>cSoundChannel</b>	Класс, используемый для воспроизведения отдельного звука. Вы можете использовать одновременно до 32 таких классов (это значит, что вы одновременно можете воспроизводить 32 звука)!
<b>cMusicChannel</b>	Класс используется для воспроизведения одного файла песни в формате MIDI или родном формате DirectMusic. Одновременно можно использовать только один такой класс.
<b>cDLS</b>	Объект класса загружаемых звуков (Downloadable Sound). Этот класс позволяет вам загрузить свои инструменты в объект <b>cMusicChannel</b> .
<b>cMP3</b>	Объект класса воспроизведения MP3-музыки. Класс позволяет воспроизводить песни в формате MP3 и определять состояние процесса воспроизведения.

Давайте двинемся вперед и начнем с верха списка, взглянув сначала на объект **cSound**.

## Управление DirectX Audio с cSound

Объект **cSound** управляет объектами DirectSound и DirectMusic и регулирует общую громкость воспроизведения. Он также управляет потоком уведомлений для потокового воспроизведения звуков. Взгляните на объявление класса:

```
class cSound
{
protected:
    HWND m_hWnd;    // Дескриптор родительского окна
    long m_Volume;  // Общая громкость

    // События для каждого звукового канала,
    // дополнительное событие используется для
    // выключения потока
    HANDLE m_Events[33];
    cSoundChannel *m_EventChannel[32];

    // Данные потока потокового воспроизведения
    HANDLE m_hThread;    // Дескриптор потока
    DWORD m_ThreadID;    // ID потока
    BOOL m_ThreadActive; // Активность потока
    static DWORD HandleNotifications(LPVOID lpvoid);

    // COM-объекты DirectSound
    IDirectSound8 *m_pDS;
```

```

IDirectSoundBuffer *m_pDSBPrimary;

// Уровень кооперации, частота,
// количество каналов и размер выборки
long m_CooperativeLevel;
long m_Frequency;
short m_Channels;
short m_BitsPerSample;

// COM-объекты DirectMusic
IDirectMusicPerformance8 *m_pDMPPerformance;
IDirectMusicLoader8 *m_pDMLoader;

public:
    cSound(); // Конструктор
    ~cSound(); // Деструктор

    // Назначение и освобождение событий, используемых
    // при потоковом воспроизведении
    BOOL AssignEvent(cSoundChannel *Channel,
                    short *EventNum, HANDLE *EventHandle);
    BOOL ReleaseEvent(cSoundChannel *Channel, short *EventNum);

    // Функции для получения COM-интерфейсов
    IDirectSound8 *GetDirectSoundCOM();
    IDirectSoundBuffer *GetPrimaryBufferCOM();
    IDirectMusicPerformance8 *GetPerformanceCOM();
    IDirectMusicLoader8 *GetLoaderCOM();

    // Функции инициализации и завершения работы
    BOOL Init(HWND hWnd, long Frequency = 22050,
              short Channels = 1, short BitsPerSample = 16,
              long CooperativeLevel = DSSCL_PRIORITY);
    BOOL Shutdown();

    // Получение и установка общего уровня громкости
    long GetVolume();
    BOOL SetVolume(long Percent);

    // Восстановление системы в исходное состояние
    BOOL Restore();
};

```

Основные функции класса **cSound**, с которыми вы будете иметь дело, — это **Init**, **Shutdown** и **SetVolume**. Я уже упоминал, что каждый объект класса перед использованием нуждается в инициализации, и класс **cSound** в этом отношении не исключение.

Для использования **Init** вы должны передать ей дескриптор родительского окна, а также необязательные параметры микширования (по умолчанию система устанавливает частоту дискретизации 22 050 Гц, монофонический звук, 16-разрядную выборку с уровнем кооперации **DSSCL\_PRIORITY**). За информацией о различных форматах воспроизведения и уровнях кооперации, которые вы можете использовать, обращайтесь к главе 4, «Воспроизведение звуков с DirectX Audio и DirectShow». За вызовом **Init** всегда должен следовать вызов **Shutdown**, делаемый когда вы завершаете работу со звуковой системой.



Для изменения уровня громкости вызовите **cSound::SetVolume** с параметром **Percent**, которому присвоено значение из диапазона от 0 (тишина) до 100 (полная громкость).

## Использование волновых данных и cSoundData

Вы используете объект класса **cSoundData** для хранения отдельного звука (волновых данных) и его описания. В объявление класса помещены данные о частоте дискретизации звука, размере выборки, количестве каналов, размере и источнике:

```
class cSoundData
{
    friend class cSoundChannel; // Пусть у звукового канала
                                // будет доступ к моему классу

protected:
    long m_Frequency;           // Частота дискретизации
    short m_Channels;           // Количество каналов
    short m_BitsPerSample;      // Размер выборки
    FILE *m_fp;                 // Указатель на файл-источник звука
    char *m_Ptr;                // Указатель на источник звука в памяти
    char *m_Buf;                // Буфер исходного звука
    long m_Size;                // Размер звука (в байтах)
    long m_Left;                // Оставшиеся данные потока
    long m_StartPos;            // Начальная позиция звука в источнике
    long m_Pos;                 // Текущая позиция звука в источнике

public:
    cSoundData();               // Конструктор
    ~cSoundData();              // Деструктор

    char *GetPtr();             // Получение указателя на
                                // звуковой буфер в памяти
    long GetSize();             // Получение размера звука

    BOOL Create();               // Создание звука, используя
                                // загруженный размер
    BOOL Create(long Size);      // Создание звука с указанием размера
    BOOL Free();                // Освобождение звукового буфера

    // Установка формата воспроизведения загруженного звука
    BOOL SetFormat(long Frequency, short Channels, short BitsPerSample);

    // Установка файла или объекта в памяти в качестве источника
    // с указанием смещения начальной позиции и длины звука
    BOOL SetSource(FILE *fp, long Pos = -1, long Size = -1);
    BOOL SetSource(void *Ptr, long Pos = -1, long Size = -1);

    // Загрузка WAV-файла в память и настройка воспроизведения
    BOOL LoadWAV(char *Filename, FILE *fp = NULL);

    // Загрузка заголовка WAV-файла и конфигурирование формата
    BOOL LoadWAVHeader(char *Filename, FILE *fp = NULL);

    // Копирование внутренних данных в другой объект cSoundData
    BOOL Copy(cSoundData *Source);
};
```

Рассматриваемый далее объект **cSoundChanel** использует класс **cSoundData** для воспроизведения звука. Однако, перед тем, как вы сможете воспроизводить звук, необходимо использовать объект класса **cSoundData** для сохранения формата воспроизведения и источника звуковых данных. Звуки могут поступать из двух источников: из файла или из буфера в памяти. Кроме того, для звуков, которые слишком велики, чтобы поместиться в память, можно сконфигурировать потоковое воспроизведение источника.

Самый быстрый способ загрузить отдельный WAV-файл — использование функции **LoadWAV**. У нее два параметра: имя загружаемого файла и указатель на файл-источник. Использовать надо только один параметр, а для другого указывать **NULL**. Указатель на файл-источник позволяет упаковать несколько WAV-файлов в один файл и при этом иметь возможность, загружать их по отдельности.

Чтобы загрузить отдельный WAV-файл, используйте следующий код:

```
cSoundData Data;  
  
// Загрузка звука из файла  
Data.LoadWAV("sound.wav");  
  
// Загрузка звука через указатель на файл  
FILE *fp = fopen("sound.wav", "rb");  
Data.LoadWAV(NULL, fp);  
fclose(fp);
```

Помимо загрузки отдельного WAV-файла у вас есть возможность установить источник ваших звуковых данных. Это полезно, если ваши звуки слишком велики для звукового буфера (больше 64 Кбайт). Вы можете использовать потоковое воспроизведение звука из файла или из буфера в памяти. Это цель функции **SoundData::SetSource**, у которой есть две версии:

```
BOOL cSoundData::SetSource(FILE *fp, long Pos = -1,  
                           long Size = -1);  
BOOL cSoundData::SetSource(void *Ptr, long Pos = -1,  
                           long Size = -1);
```

Как видите, можно выбирать, передать ли указатель на файл-источник или указатель на буфер в памяти. Параметр **Pos** передает классу местоположение начала звуковых данных (смещение). Параметр **Size** задает общее количество байт в потоке (размер звука).

---

**ВНИМАНИЕ!**

Когда вы используете буфер в памяти, вы должны всегда освобождать его вызовом **cSoundData::Free**. Также помните, что если вы выполняете потоковое воспроизведение звука из файла, ответственность за закрытие файла лежит на вас.

---

Обратите внимание, что по умолчанию аргументам **Pos** и **Size** присваивается значение **-1**, что позволяет классу самому установить эти

значения. Чтобы это произошло, вы должны сперва установить формат воспроизведения с помощью функции **SetFormat**, которая является самодокументируемой. Затем вы должны проанализировать заголовок волнового файла, воспользовавшись функцией **LoadWAVHeader**, в отношении параметров аналогичной **LoadWAV**.

И, наконец, если звук будет храниться в памяти, вы должны создать буфер с помощью функции **cSoundData::Create**. Вы можете сами указать размер буфера, или позволить функции использовать размер буфера, вычисленный **LoadWAVHeader**. Вызов **cSoundData::GetPtr** возвращает указатель на буфер, который вы безопасно можете использовать для хранения звука.

В качестве примера предположим, что вы хотите воспроизвести большой WAV-файл с именем **BigSound.wav**. Для инициализации класса **cSoundData** можно использовать следующий код:

```
cSoundData Data;

FILE *fp = fopen("BigSound.wav", "rb");

Data.LoadWAVHeader(NULL, fp); // Получаем параметры воспроизведения
Data.SetSource(fp);          // Устанавливаем файл источника

// Воспроизводим звук и по завершении закрываем файл
fclose(fp);
```

## Воспроизведение звука с cSoundChannel

Сейчас у вас есть инициализированная звуковая система и загруженные звуковые данные. Вполне естественно перейти к воспроизведению звука. Это назначение класса **cSoundChannel**, объявление которого показано ниже.

```
// Фиксированные размеры буферов звукового канала
const long g_SoundBufferSize = 65536;
const long g_SoundBufferChunk = g_SoundBufferSize / 4;
```

Перед тем, как углубляться в код, я хочу поговорить о двух глобальных константах. Первая, **g\_SoundBufferSize**, представляет количество байтов, выделяемых для каждого буфера DirectSound, используемого при воспроизведении звука. Я использую 65 536 байт, чего достаточно для хранения нескольких секунд воспроизводимых данных даже в форматах с высоким качеством.

Вторая переменная, **g\_SoundBufferChunk**, это размер отдельного фрагмента, которых всего четыре. Каждый фрагмент содержит небольшую выборку потокового звука. Когда воспроизведение фрагмента завершено, начинается воспроизведение следующего в списке фрагмента, в то время как в предыдущий фрагмент загружаются новые звуковые данные.

Не следует менять эти значения, только если вы не хотите сэкономить память; в этом случае измените значение **g\_SoundBufferSize** на меньшее. Есть несколько преимуществ наличия звуковых буферов большего

размера — чем больше размер, тем реже ядро должно подгружать в поток новые данные. Конечно, это означает больший расход памяти, но что такое для современной системы с сотнями мегабайт памяти, пара мегабайт, потраченные для звуковых данных?

Давайте вернемся к объявлению класса **cSoundChannel**:

```
class cSoundChannel
{
    friend class cSound; // Предоставить доступ классу cSound

protected:
    cSound *m_Sound;           // Родительский класс cSound
    IDirectSoundBuffer8 *m_pDSBuffer; // Звуковой буфер DS
    IDirectSoundNotify8 *m_pDSNotify; // Объект уведомления

    short      m_Event; // Количество событий для уведомлений
    long       m_Volume; // Текущая громкость 0-100%
    signed long m_Pan;   // Позиционирование от -100 до +100
    BOOL       m_Playing; // Флаг воспроизведения канала
    long       m_Loop;   // Количество повторов воспроизведения

    long m_Frequency; // Формат воспроизведения
    short m_BitsPerSample; // для инициализированного
    short m_Channels;     // канала
    cSoundData m_Desc;    // Описание источника звука

    // Переменные для потокового воспроизведения
    short m_LoadSection; // Следующий загружаемый фрагмент
    short m_StopSection; // На каком фрагменте остановиться
    short m_NextNotify;  // Какой фрагмент следующий

    BOOL BufferData(); // Входной буфер потоковых данных
    BOOL Update();    // Обновление воспроизведения канала

public:
    cSoundChannel(); // Конструктор
    ~cSoundChannel(); // Деструктор

    // Функции для получения COM-объектов
    IDirectSoundBuffer8 *GetSoundBufferCOM();
    IDirectSoundNotify8 *GetNotifyCOM();

    // Создание и освобождение звукового канала
    BOOL Create(cSound *Sound, long Frequency = 22050,
               short Channels = 1, short BitsPerSample = 16);
    BOOL Create(cSound *Sound, cSoundData *SoundDesc);
    BOOL Free();

    // Воспроизведение и остановка канала
    BOOL Play(cSoundData *Desc, long VolumePercent = 100,
              long Loop = 1);
    BOOL Stop();

    // Получение и установка уровня громкости (0-100%)
    long GetVolume();
    BOOL SetVolume(long Percent);

    // Получение и установка позиционирования
    // (от -100 слева до +100 справа)
```

```
signed long GetPan();
BOOL SetPan(signed long Level);

// Получение и установка частоты воспроизведения
long GetFrequency();
BOOL SetFrequency(long Level);

BOOL IsPlaying(); // Возвращает TRUE если звук воспроизводится
};
```

На фоне простого класса **cSoundData** размер класса **cSoundChannel** впечатляет. Вы можете создать 32 экземпляра этого класса, а это значит, что вы одновременно можете воспроизводить 32 канала (звуковое ядро не позволяет иметь одновременно более 32 экземпляров класса — инициализация класса после первых 32 будет вызывать ошибку). Каждый звуковой канал инициализируется вызовом **cSoundChannel::Create**.

При вызове **Create** вы предоставляете ранее инициализированный класс **cSound** и формат воспроизведения. Чтобы жизнь была легче, вы можете создать звуковой канал, используя формат воспроизведения, хранящийся в классе **cSoundData**. Когда вы завершите работу с классом **cSoundChannel**, освободите его ресурсы, вызвав **cSoundChannel::Free**.

Скорее всего, вы будете работать с **cSoundChannel** для воспроизведения и остановки звуков и, возможно, будете менять их громкость. Для воспроизведения звука передайте **cSoundChannel** объект **cSoundData**, содержащий требуемый звук, вместе с уровнем громкости и количеством повторов воспроизведения. Чтобы звук воспроизводился в бесконечном цикле, укажите в параметре **Loop** значение 0.

Остальные функции самодocumented. Для уровня громкости и позиционирования используются процентные значения от -100% (тишина или крайняя левая позиция) до +100% (полная громкость или крайняя правая позиция). Вызов **cSoundChannel::IsPlaying** возвращает **TRUE**, если звук воспроизводится, и **FALSE** — если нет.

Вот пример, который загружает отдельный звук и запускает потоковое воспроизведение большого звука, используя два звуковых канала:

```
// Глобальные объявления
cSound      g_Sound;
cSoundData  g_Data[2];
cSoundChannel g_Channel[2];

// Инициализация звуковой системы
// Подразумеваем, что hWnd уже содержит
// инициализированный дескриптор окна
g_Sound.Init(hWnd);

// Загрузка звуков
g_Data[0].LoadWAV("SmallSound.wav");
FILE *fp = fopen("BigSound.wav", "rb");
```

```

g_Data[1].LoadWAVHeader(NULL, fp);
g_Data[1].SetSource(fp);

// Создание звуковых каналов
g_Channels[0].Create(&g_Sound, &g_Data[0]);
g_Channels[1].Create(&g_Sound, &g_Data[1]);

// Начало воспроизведения
g_Channels[0].Play(&g_Data[0]);           // Один раз воспроизводим
                                           // первый звук
g_Channels[1].Play(&g_Data[1], 100, 0);   // Воспроизводим второй звук
                                           // в бесконечном цикле

// Когда все готово, останавливаем все и завершаем работу
g_Channels[0].Stop();
g_Channels[0].Free();
g_Channels[1].Stop();
g_Channels[1].Free();
g_Data[0].Free();
g_Data[1].Free();
fclose(fp);
g_Sound.Shutdown();

```

## Слушаем музыку с cMusicChannel

Повторю еще раз, что за удовольствие от игры без музыки? Пришло время нанести удар с помощью класса **cMusicChannel**, который воспроизводит MIDI-файлы и песни в родном формате DirectMusic (\*.SGT):

```

class cMusicChannel
{
    friend class cSound; // Разрешаем классу cSound доступ к данным
protected:
    cSound *m_Sound;           // Родительский класс cSound
    IDirectMusicSegment8 *m_pDMSegment; // Объект сегмента DM
    long m_Volume;             // Уровень громкости 0-100%
public:
    cMusicChannel(); // Конструктор
    ~cMusicChannel(); // Деструктор

    IDirectMusicSegment8 *GetSegmentCOM(); // Получить COM сегмента

    BOOL Create(cSound *Sound); // Инициализация класса
    BOOL Load(char *Filename); // Загрузка музыкального файла
    BOOL Free(); // Освобождение музыкального файла
    BOOL SetDLS(cDLS *DLS); // Установка нового DLS

    // Воспроизведение и остановка музыки
    BOOL Play(long VolumePercent = 100, long Loop = 1);
    BOOL Stop();

    // Получение и установка уровня громкости (0-100%)
    long GetVolume();
    BOOL SetVolume(long Percent = 100);

    BOOL SetTempo(long Percent = 100); // Установка темпа

    BOOL IsPlaying(); // TRUE если песня воспроизводится,
                     // FALSE - если нет
};

```

Пусть размер класса **cMusicChannel** не обманывает вас — он выполняет всю необходимую работу. Отличие **cMusicChannel** от других классов заключается в том, что его нужно инициализировать только один раз вызовом **cMusicChannel::Create**.

Функция **cMusicChannel::Free** выгружает песню из памяти, освобождая место для загрузки следующей песни. Загружая песню вы указываете имя файла, который должен быть MIDI-файлом или песней в родном формате DirectMusic (.SGT). У MIDI-файла должно быть расширение .MID, иначе функции **cMusicChannel** не смогут сконфигурировать DirectMusic для правильного воспроизведения. Если вы используете только MIDI-файлы, то можете изменить функции, чтобы всегда выполнялось принудительное конфигурирование объекта сегмента песни DirectMusic для воспроизведения MIDI (как было показано в главе 4).

Когда песня загружена, вы можете начать воспроизведение, используя функцию **cMusicChannel::Play**, параметры **Volume** и **Loop** которой работают точно так же, как одноименные параметры **cSoundChannel::Play**. Оставшиеся функции просты для понимания — за исключением **cMusicChannel::SetDLS**, которая меняет инструменты, используемые для воспроизведения.

Я доберусь до материала о DLS в следующем разделе («Смешивание инструментов с cDLS»), а сейчас взгляните на класс **cMusicChannel** в действии:

```
// Глобальные объявления
cSound      g_Sound;
cMusicChannel g_Music;

// Инициализация звуковой системы
// Подразумеваем, что hWnd уже содержит
// инициализированный дескриптор окна
g_Sound.Init(hWnd);

// Инициализация музыкального канала
g_Music.Create(&g_Sound);

// Загрузка и воспроизведение песни (бесконечный цикл)
g_Music.Load("song.mid");
g_Music.Play(100,0);

// Завершив работу останавливаем и выгружаем песню
g_Music.Stop();
g_Music.Free();

// Выключение звуковой системы
g_Sound.Shutdown();
```

## Смешивание инструментов с cDLS

Наконец-то мы добрались до конца информации об использовании звукового ядра. Чтобы улучшить возможности воспроизведения музыки класса **cMusicChannel**, показанного в предыдущем разделе, вы можете

использовать представленный здесь класс **cDLS** (обратитесь к главе 4 за описанием преимуществ, которые дает использование загружаемых звуков, называемых DLS):

```
// Макросы, помогающие работать с модификаторами
#define PATCH(m,l,p) ((m << 16) | (l << 8) | p)
#define PATCHMSB(x) ((x >> 16) & 255)
#define PATCHLSB(x) ((x >> 8) & 255)
#define PATCHNUM(x) (x & 255)

class cDLS
{
protected:
    cSound *m_Sound; // Родительский объект cSound

    // Объект DLS-коллекции DM
    IDirectMusicCollection *m_pDMCollection;

public:
    cDLS(); // Конструктор
    ~cDLS(); // Деструктор

    // Возвращает COM коллекции
    IDirectMusicCollection8 *GetCollectionCOM();

    BOOL Create(cSound *Sound); // Инициализация класса

    // Загрузка и освобождение DLS
    // (NULL = загрузка набора по умолчанию)
    BOOL Load(char *Filename = NULL);
    BOOL Free();

    long GetNumPatches(); // Возвращает количество
                        // модификаторов в наборе
    long GetPatch(long Index); // Возвращает модификатор
                        // с указанным номером
    BOOL Exists(long Patch); // Проверяет, существует ли
                        // модификатор с заданным номером
};
```

Как видите, единственная цель класса **cDLS** — хранить отдельный набор DLS. Как и в случае с **cMusicChannel**, вы вызываете **cDLS::Create** только один раз, поскольку **cDLS::Free** освобождает только загруженный набор. Обратите внимание, что для параметра **Filename** в **cDLS::Load** по умолчанию устанавливается значение **NULL**, указывающее, что надо загрузить используемый по умолчанию набор DLS. Загрузка используемого по умолчанию набора DLS очень удобна при восстановлении оригинального звучания инструментов.

Последние три функции показывают инструменты, которые содержит набор DLS. Это также назначение макросов **PATCH**, находящихся в начале объявления класса. Чтобы увидеть, сколько инструментов содержится в классе, вызовите **cDLS::GetNumPatches**.

Теперь вы можете в цикле перебрать все инструменты для получения их номеров модификаторов с помощью функции **cDLS::GetPatch**, или,



воспользовавшись **cDLS::Exist**, проверить существует ли указанный модификатор в наборе. Функция возвращает **TRUE**, если модификатор существует, и **FALSE**, если нет.

При использовании **cDLS** с **cMusicChannel** вы загружаете требуемый DLS и вызываете **cMusicChannel::SetDLS** для использования этого набора инструментов:

```
// Предполагается наличие ранее загруженного объекта c_Music
// и ранее инициализированного объекта g_Sound
cDLS g_DLS;

g_DLS.Create(&g_Sound);
g_DLS.Load("custom.dls");

g_Music.SetDLS(&g_DLS);

// Завершив работу с DLS, освободите его
g_DLS.Free();
```

## Воспроизведение MP3 с cMP3

В главе 4 вы увидели, как просто воспроизводить MP3-файлы, используя DirectShow. Настало время применить знания в работе и создать класс для воспроизведения MP3. В этом классе, названном **cMP3**, вы можете инициализировать и выключать DirectShow, визуализировать медиа-файл и управлять воспроизведением песни — включая, выключая и приостанавливая ее, — как вам угодно. Все это находится в **cMP3**:

```
class cMP3
{
private:
    IGraphBuilder *m_pDSGraphBuilder;
    IMediaControl *m_pDSMediaControl;
    IMediaEvent *m_pDSMediaEvent;

public:
    cMP3();
    ~cMP3();

    // Инициализация и выключение DirectShow
    BOOL Init();
    BOOL Shutdown();

    // Визуализация файла для использования
    BOOL Render(char *Filename);

    // Управление воспроизведением
    BOOL Play();
    BOOL Stop();
    BOOL Pause();

    // Состояние воспроизведения
    // (TRUE = идет воспроизведение)
    BOOL Playing();
};
```

Чтобы применить предоставляемую данным классом возможность воспроизведения MP3, воспользуйтесь следующим фрагментом кода:

```
сMP3 MP3;

// Инициализация объекта класса
MP3.Init();

// Визуализация файла и начало воспроизведения
if(MP3.Render("song.mp3") == TRUE)
    MP3.Play();

// Ждем завершения песни
while(MP3.Playing() == TRUE);

// Выключаем все
MP3.Shutdown();
```

## Сетевое ядро

В предыдущей главе вы увидели, как просто использовать DirectPlay. Теперь вы узнаете, как работать с DirectPlay, используя сетевое ядро. Сетевое ядро содержит три класса: **cNetworkAdapter**, **cNetworkServer** и **cNetworkClient**.

### Запрос адаптера с cNetworkAdapter

Вы используете **cNetworkAdapter** для перечисления установленных в вашей системе TCP/IP-устройств. Чтобы установить соединение вам необходимо знать GUID устройства, и его получение является целью класса **cNetworkAdapter**. Вот как выглядит объявление класса:

```
class cNetworkAdapter
{
protected:
    DPN_SERVICE_PROVIDER_INFO *m_AdapterList; // Список адаптеров
    unsigned long m_NumAdapters;             // Количество адаптеров

    // Пустой обработчик сетевых сообщений - необходим
    static HRESULT WINAPI NetMsgHandler(
        PVOID pvUserContext, DWORD dwMessageId,
        PVOID pMsgBuffer) { return S_OK; }

public:
    cNetworkAdapter(); // Конструктор
    ~cNetworkAdapter(); // Деструктор

    BOOL Init(); // Инициализация объекта класса
    BOOL Shutdown(); // Завершение работы объекта
                    // (освобождение памяти)

    long GetNumAdapters(); // Получение количества
                           // установленных адаптеров

    // Сохранение имени адаптера в буфере
    // (Num от 0 до количества адаптеров минус 1)
    BOOL GetName(unsigned long Num, char *Buf);
```

```
// Возвращает указатель на GUID адаптера
// (Num от 0 до количества адаптеров минус 1)
GUID *GetGUID(unsigned long Num);
};
```

Использовать класс **cNetworkAdapter** просто: вызовите функцию **Init**, запросите количество установленных адаптеров, а затем начинайте извлекать имена адаптеров и их GUID. Когда завершите работу с объектом, вызовите **Shutdown** для освобождения внутренних ресурсов класса.

Я вернусь к использованию класса **cNetworkAdapter** в следующих двух подразделах. А сейчас давайте перейдем к классу **cNetworkServer**.

## Серверы и cNetworkServer

На серверной стороне сети вы имеете дело с классом **cNetworkServer**, который позволяет инициализировать серверные объекты **DirectPlay**, открыть игровую сессию и обрабатывать входящие и исходящие сетевые сообщения. В данном разделе вы увидите, как я оборачиваю операции серверной стороны в представленный ниже класс **cNetworkServer**:

```
class cNetworkServer
{
protected:
    IDirectPlay8Server *m_pDPServer; // Объект сервера

    BOOL m_Connected;                // Флаг запуска узла

    // Имя и пароль сессии (хранятся как ASCII-символы)
    char m_SessionName[MAX_PATH];
    char m_SessionPassword[MAX_PATH];

    long m_Port;                     // Используемый порт
    long m_MaxPlayers;               // Максимально допустимое
                                    // количество игроков
    long m_NumPlayers;               // Текущее количество игроков

    // Обработчик сетевых сообщений
    static HRESULT WINAPI NetworkMessageHandler(
        PVOID pvUserContext, DWORD dwMessageId,
        PVOID pMsgBuffer);

    // Перегружаемые функции для различных
    // сетевых сообщений
    virtual BOOL AddPlayerToGroup(
        DPNMSG_ADD_PLAYER_TO_GROUP *Msg) { return TRUE; }
    virtual BOOL AsyncOpComplete(
        DPNMSG_ASYNC_OP_COMPLETE *Msg) { return TRUE; }
    virtual BOOL ClientInfo(
        DPNMSG_CLIENT_INFO *Msg) { return TRUE; }
    virtual BOOL ConnectComplete(
        DPNMSG_CONNECT_COMPLETE *Msg) { return TRUE; }
    virtual BOOL CreateGroup(
        DPNMSG_CREATE_GROUP *Msg) { return TRUE; }
    virtual BOOL CreatePlayer(
        DPNMSG_CREATE_PLAYER *Msg) { return TRUE; }
    virtual BOOL DestroyGroup(
        DPNMSG_DESTROY_GROUP *Msg) { return TRUE; }
```

```

virtual BOOL DestroyPlayer(
    DPNMSG_DESTROY_PLAYER *Msg) { return TRUE; }
virtual BOOL EnumHostsQuery(
    DPNMSG_ENUM_HOSTS_QUERY *Msg) { return TRUE; }
virtual BOOL EnumHostsResponse(
    DPNMSG_ENUM_HOSTS_RESPONSE *Msg) { return TRUE; }
virtual BOOL GroupInfo(
    DPNMSG_GROUP_INFO *Msg) { return TRUE; }
virtual BOOL HostMigrate(
    DPNMSG_HOST_MIGRATE *Msg) { return TRUE; }
virtual BOOL IndicateConnect(
    DPNMSG_INDICATE_CONNECT *Msg) { return TRUE; }
virtual BOOL IndicatedConnectAborted(
    DPNMSG_INDICATED_CONNECT_ABORTED *Msg) { return TRUE; }
virtual BOOL PeerInfo(
    DPNMSG_PEER_INFO *Msg) { return TRUE; }
virtual BOOL Receive(
    DPNMSG_RECEIVE *Msg) { return TRUE; }
virtual BOOL RemovePlayerFromGroup(
    DPNMSG_REMOVE_PLAYER_FROM_GROUP *Msg) { return TRUE; }
virtual BOOL ReturnBuffer(
    DPNMSG_RETURN_BUFFER *Msg) { return TRUE; }
virtual BOOL SendComplete(
    DPNMSG_SEND_COMPLETE *Msg) { return TRUE; }
virtual BOOL ServerInfo(
    DPNMSG_SERVER_INFO *Msg) { return TRUE; }
virtual BOOL TerminateSession(
    DPNMSG_TERMINATE_SESSION *Msg) { return TRUE; }

public:
    cNetworkServer(); // Конструктор
    ~cNetworkServer(); // Деструктор

    IDirectPlay8Server *GetServerCOM(); // Возвращает объект сервера

    BOOL Init(); // Инициализирует сетевой сервер
    BOOL Shutdown(); // Выключает сетевой сервер

    // Начинает сессию на узле
    BOOL Host(GUID *guidAdapter, long Port,
        char *SessionName, char *Password = NULL,
        long MaxPlayers = 0);
    BOOL Disconnect(); // Завершение сессии
    BOOL IsConnected(); // Проверяет, запущена ли сессия

    // Передача необработанных данных или текстовой строки
    BOOL Send(DPNID dpnidPlayer, void *Data,
        unsigned long Size, unsigned long Flags=0);
    BOOL SendText(DPNID dpnidPlayer, char *Text,
        unsigned long Flags=0);

    // Принудительное отключение игрока
    BOOL DisconnectPlayer(long PlayerId);

    // Получение IP-адреса игрока или сервера в указанный буфер
    BOOL GetIP(char *IPAddress, unsigned long PlayerId = 0);

    // Получение имени игрока
    BOOL GetName(char *Name, unsigned long PlayerId);

    // Получение номера используемого порта
    long GetPort();

```

```
// Иолучение имени сессии и пароля
BOOL GetSessionName(char *Buf);
BOOL GetSessionPassword(char *Buf);

// Получение максимально возможного и текущего
// количества игроков
long GetMaxPlayers();
long GetNumPlayers();
};
```

Для того, чтобы использовать класс **cNetworkServer** (точно так же как и класс **cNetworkClient**, о котором мы поговорим в следующем разделе), вы наследуете собственный класс, используя **cNetworkServer** в качестве базового. Это необходимо потому что требуется перегрузка обработчиков сообщений вашими собственными функциями. В классе **cNetworkServer** представлены все сетевые сообщения, так что наследуемый класс не пропустит важной информации.

Чтобы начать сессию вам необходим GUID адаптера, имя сессии, необязательный пароль и максимально возможное количество игроков (0 означает, что ограничений нет). При вызове **cNetworkServer::Host DirectPlay** инициализирует подключения и возвращает управление вам. После этого вы можете начать ожидание входящих сообщений. Ваша работа — закачивать входящие сообщения и поступать с ними так, как считаете нужным. Создаваемые вами обработчики сообщений должны возвращать **TRUE**, если сообщение успешно обработано, и **FALSE**, если произошла ошибка.

В качестве примера здесь приведен экземпляр класса **cNetworkServer**, который отображает текст входящего сообщения и отправляет точно такое же сообщение назад отправителю (используя гарантированную доставку):

```
// Создаем класс-наследник
class cServer : public cNetworkServer
{
private:
    BOOL Receive(DPNMSG_RECEIVE *Msg);
};

BOOL cServer::Receive(DPNMSG_RECEIVE *Msg)
{
    // Отображаем сообщение
    MessageBox(NULL, Msg->pReceivedData,
        "Incoming Message", MB_OK);

    // Отправляем его обратно
    Send(Msg->dpnidSender, Msg->pReceiveData,
        Msg->dwReceivedDataSize, DPNSSEND_GUARANTEED);

    return TRUE;
}
```

```

int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hPrev,
                  LPSTR szCmdLine, int nCmdShow)
{
    cServer          Server;
    cNetworkAdapter Adapter;
    GUID             *guidAdapter;

    // Выбираем первый сетевой адаптер
    Adapter.Init();
    guidAdapter = Adapter.GetGUID(0); // 0 = первый адаптер
    Server.Init();
    Server.Host(guidAdapter, 12345, "TextSession");

    // Ждем нажатия ESC
    while(!(GetAsyncKeyState(VK_ESC) & 0x80));

    Server.Disconnect();
    Server.Shutdown();

    // Освобождаем список адаптеров
    Adapter.Shutdown();

    return TRUE;
}

```

Не беспокойтесь, если вам показалось, что приведенный пример показывает слишком мало; вы ближе познакомитесь с сетевыми компонентами в главе 15, «Сетевой режим для многопользовательской игры».

## Работа клиента и cNetworkClient

Пришло время взглянуть на последний сетевой класс, **cNetworkClient**, который работает на клиентской стороне сети:

```

class cNetworkClient
{
protected:
    IDirectPlay8Client *m_pDPClient; // Объект клиента DP

    BOOL m_Connected; // Флаг подключения

    char m_IPAddress[MAX_PATH]; // IP-адрес
    long m_Port;                // Порт подключения
    char m_Name[MAX_PATH];      // Имя клиента

    // Имя и пароль сессии (пересылается серверу)
    char m_SessionName[MAX_PATH];
    char m_SessionPassword[MAX_PATH];

    // Функция обработки сетевых сообщений
    static HRESULT WINAPI NetworkMessageHandler(
        PVOID pvUserContext, DWORD dwMessageId,
        PVOID pMsgBuffer);

    // Далее идут перегружаемые обработчики сетевых
    // сообщений (такие же, как и в cNetworkServer).
    // Здесь они не приводятся для экономии места

```

```
public:
    cNetworkClient(); // Конструктор
    ~cNetworkClient(); // Деструктор

    // Возвращает объект клиента DirectPlay
    IDirectPlay8Client *GetClientCOM();

    BOOL Init(); // Инициализация сетевого клиента
    BOOL Shutdown(); // Выключение клиента

    // Подключение к удаленному серверу с использованием
    // указанного адаптера, IP, порта, имени игрока,
    // имени сессии, и необязательного пароля
    BOOL Connect(GUID *guidAdapter, char *IP, long Port,
                char *PlayerName, char *SessionName,
                char *SessionPassword = NULL);

    BOOL Disconnect(); // Отключение от сессии
    BOOL IsConnected(); // Возвращает TRUE при наличии подключения

    // Функции передачи необработанных данных и текста
    BOOL Send(void *Data, unsigned long Size,
              unsigned long Flags=0);
    BOOL SendText(char *Text, unsigned long Flags=0);

    BOOL GetIP(char *IPAddress); // Получение IP-адреса в буфер
    long GetPort(); // Возвращает номер порта
                        // подключения
    BOOL GetName(char *Name); // Возвращает имя
    BOOL GetSessionName(char *Buf); // Возвращает имя сессии
    BOOL GetSessionPassword(char *Buf); // Возвращает пароль
};
```

Работа с **cNetworkClient** похожа на использование **cNetworkServer**, за исключением подключения к сети. Чтобы установить подключение, используя **cNetworkClient::Connect**, вы должны сперва выбрать сетевой адаптер, используя объект класса **cNetworkAdapter**, как показано в следующем примере:

```
// Создаем класс-наследник
class cClient : public cNetworkClient
{
private:
    BOOL Receive(DPNMSG_RECEIVE *Msg);
};

BOOL cClient::Receive(DPNMSG_RECEIVE *Msg)
{
    MessageBox(NULL, Msg->pReceiveData,
                "Incoming Message", MB_OK);
    return TRUE;
}

int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hPrev,
                  LPSTR szCmdLine, int nCmdShow)
{
    cClient      Client;
    cNetworkAdapter Adapter;
    GUID         *guidAdapter;
```

```

// Выбираем первый сетевой адаптер
Adapter.Init();
guidAdapter = Adapter.GetGUID(0); // 0 = первый адаптер

// Инициализируем клиента и подключаемся к IP=123.123.123.123
// используя порт 12345, сессия = TextSession без пароля
Client.Init();
Client.Connect(guidAdapter, "123.123.123.123",
               12345, "MyName", "TextSession");

// Ждем установки соединения,
// или пока пользователь не нажмет ESC
while(Client.IsConnected() == FALSE) {
    if(GetAsyncKeyState(VK_ESC) & 0x80) {
        Client.Disconnect();
        Client.Shutdown();
        return TRUE;
    }
}

// Подключились, начинаем прикладную обработку
// Отправляем текстовое сообщение
Client.SendText("Hello there!");

// Ждем нажатия ESC
while(!(GetAsyncKeyState(VK_ESC) & 0x80));

// Разрыв соединения и выключение
Client.Disconnect();
Client.Shutdown();

// Освобождение списка адаптеров
Adapter.Shutdown();

return TRUE;
}

```

И снова пример минимален; все сетевое ядро в действии вы увидите в главе 15. Сейчас можете посмотреть на объявление класса и на код реализации, находящийся на прилагаемом к книге компакт-диске.

## Заканчиваем изучать ядро игры

Создание ядра многократно используемых объектов, таких как показанные в этой главе, — одна из лучших вещей, которыми можно заняться, начиная разрабатывать игры. Вы не только изучите внутренние детали каждого объекта и компонента, представленного классами ядра, но и создадите прочный каркас, который ускорит разработку ваших игр.

В этой главе я познакомил вас с ядром игры, набором классов, разработанных мной для использования в своих проектах и в этой книге. Я передаю это ядро вам и надеюсь, что оно окажется полезным в ваших проектах. Следуя примерам из этой главы и из оставшейся части книги вы должны получить твердое представление о том, что может делать ядро, и о том, как использовать классы ядра.



### Программы на CD-ROM

Программы, демонстрирующие обсуждавшийся в этой главе код, находятся на прилагаемом к книге CD-ROM. В папке \BookCode\Chap06\ вы найдете следующие программы:

**GameCore** — это не проект реального приложения, поскольку каталог содержит только полный исходный код ядра игры. Используйте эту коллекцию классов в качестве фундамента ваших приложений. Местоположение: \BookCode\Chap06\GameCore\.

# **Часть III**

## **Программирование ролевых игр**

**Глава 7      Использование двумерной  
графики**

**Глава 8      Создание трехмерного  
графического движка**

**Глава 9      Объединение двумерной и  
трехмерной графики**

**Глава 10     Реализация скриптов**

**Глава 11     Определение и  
использование объектов**

**Глава 12    Управление игроками и персонажами**

**Глава 13    Работа с картами и уровнями**

**Глава 14    Создание боевых последовательностей**

**Глава 15    Сетевой режим для многопользовательской игры**

# Глава 7

## Использование двухмерной графики

Наконец-то вы разобрались с основами программирования и добрались до серьезных дел. Поскольку эта книга посвящена программированию ролевых игр, давайте начнем с вещи, которую игрок замечает одной из первых — с графики! Взглянув на «вчерашний день» игр, вы увидите, что они эволюционировали от простого двухмерного графического стиля, который все еще присутствует на современном рынке в таких играх, как Final Fantasy, Baldur's Gate и различные игры серии Ultima. Но это не все — сегодня некоторые популярные игровые консоли (например, Nintendo Gameboy Advance) используют двухмерную графику. Теперь пришло время и вам использовать двухмерную графику в своих собственных играх.

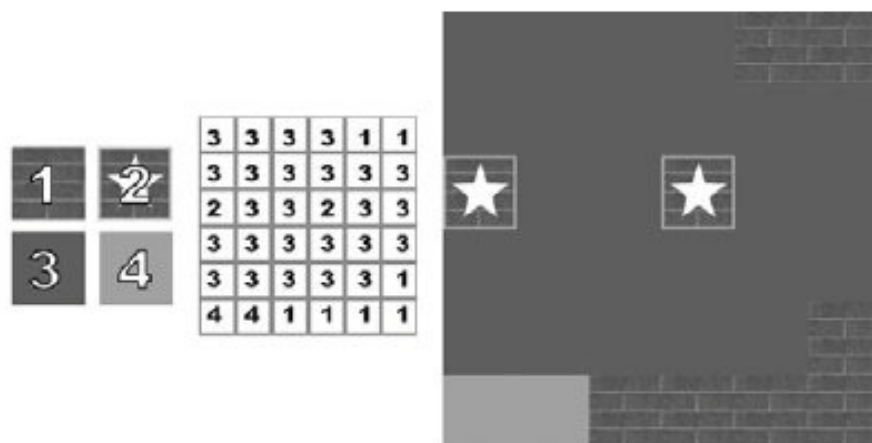
В этой главе вы узнаете следующее:

- Использование блоков и карт.
- Построение библиотеки для работы с двухмерными блоками.
- Создание библиотеки для работы с большими растрами.

### Знакомство с блоками и картами

Сердцем упомянутых во вступлении к главе двухмерных игр является техника рисования, известная как *блочная графика (tiling)*. В ней вы для конструирования большой сцены используете маленькие прямоугольные группы пикселей (эти небольшие растры называются *блоками — tiles*). Порядок размещения, определяющий как рисуются блоки, называется *картой (map)*. Все это показано на рис. 7.1.

На рис. 7.1 слева находятся четыре небольших пронумерованных изображения — это блоки. В середине вы видите таблицу с числами. Эта таблица задает порядок, в котором рисуются блоки (в стиле «раскрась по номерам цветов»). Для каждого элемента таблицы вы рисуете соответствующий блок, который представлен числом и продолжаете это, пока не нарисуете всю сцену. Таблица называется картой. После завершения обработки таблицы получается изображение, показанное справа.



**Рис. 7.1.** В рисовании больших двумерных сцен блоки и карты идут рука об руку. Вы используете изображенные слева блоки для визуализации карты справа

Полезь от использования блоков и карт ясна — они требуют очень мало памяти для хранения. Используя для карты формат «раскрась по номерам» вы сможете конструировать огромные (именно *огромные*) сцены, тратя на них очень мало памяти.

Предположим, к примеру, что используются блоки размером  $16 \times 16$  пикселей. Предположим, что вы используете 8-разрядный цвет, тогда для набора из 256 блоков потребуется 65 536 байт памяти. Кроме того, у вас есть карта, представляющая собой массив байтов (каждый байт представляет номер рисуемого блока) размером  $1024 \times 1024$  (1 048 576 байт). Итак, всего для хранения карты и блоков вам требуется 1 114 112 байт.

Поскольку размер каждого блока равен 16 пикселям, а размер карты — 1024 блока, в результате визуализации получается изображение размером  $16\,384 \times 16\,384$  пиксела, для хранения которого требуется 268 435 456 байт. Вы видите как экономится память вместо почти 300 Мбайт чуть больше 1 Мбайт.

#### ПРИМЕЧАНИЕ

Здесь мы сделали краткий обзор основ использования блоков и карт. Вы могли не понять насколько далеко простирается концепция использования блоков. Несмотря на то, что вы рисуете прямоугольные блоки, сама графика не должна быть прямоугольной, и даже не должна рисоваться на экране в виде прямоугольной решетки.

## Использование блоков с DirectX

Как вы уже видели в предыдущей главе, нарисовать небольшой прямоугольный текстурированный полигон, который замечательно подойдет для представления ваших блоков, совсем не сложно. Вы можете выполнить это с помощью специального объекта D3DX с именем **ID3DXSprite**. В главе 6, «Создаем ядро игры», я упоминал интерфейс **ID3DXSprite**, когда обсуждал графическое ядро; теперь у вас есть шанс поближе познакомиться с этим интерфейсом.

У объекта **ID3DXSprite** одна задача — рисовать на экране прямоугольные полигоны, используя назначенную вами текстуру. Конечно же, в предоставленной текстуре будут упакованы блоки.

Чтобы начать использовать блоки в Direct3D, объявите экземпляр объекта **ID3DXSprite** и используйте для его инициализации функцию **D3DXCreateSprite**:

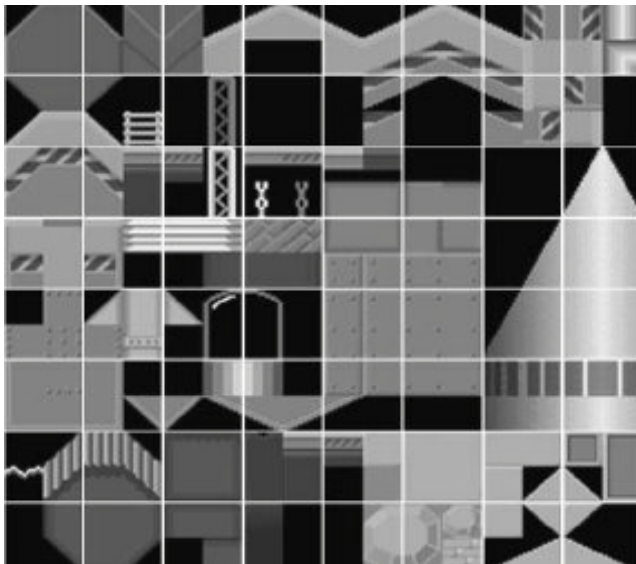
```
// g_pD3DDevice = ранее инициализированный объект устройства
ID3DXSprite *pSprite = NULL;

if(FAILED(D3DXCreateSprite(g_pD3DDevice, &pSprite))) {
    // Произошла ошибка
}
```

Как видите, функция **D3DXCreateSprite** получает два параметра — указатель на ранее инициализированный объект трехмерного устройства и указатель на создаваемый объект **ID3DXSprite**. После этого вызова функции все готово к работе. Вам необходимо только загрузить текстуру, представляющую блок (или блоки), который вы собираетесь рисовать и воспользоваться функцией **ID3DXSprite::Draw** для рисования блока:

```
HRESULT ID3DXSprite::Draw(
    IDirect3DTexture9 *pSrcTexture,      // Используемая текстура
    CONST RECT        *pSrcRect,         // Прямоугольник источника
    CONST D3DXVECTOR2 *pScaling,         // Вектор масштабирования
    CONST D3DXVECTOR2 *pRotationCenter,  // Центр вращения
    FLOAT             Rotation,          // Угол поворота
    CONST D3DVECTOR2  *pTranslation,     // Вектор перемещения
    D3DCOLOR          Color);            // Модуляция цвета
```

Трюк в использовании функции **Draw** заключается в конструировании структуры **RECT** для прямоугольника источника, содержащей координаты фрагмента внутри текстуры, который вы хотите использовать как блок. Например, у вас есть текстура, размером  $256 \times 256$  пикселей, содержащая 64 блока (каждый размером  $32 \times 32$  пикселя), упакованных в 8 строк и 8 столбцов (как показано на рис. 7.2).



*Рис. 7.2. 64 блока, упорядоченные в таблицу  $8 \times 8$*

**СОВЕТ**

Чтобы повысить эффективность, упаковывайте в используемую для блоков текстуру столько блоков, сколько можете поместить и располагайте их в виде строк и столбцов. Например, если у вас 64 блока размером  $32 \times 32$  пиксела, создайте растровое изображение для хранения 8 строк и 8 столбцов блоков, что дает вам текстуру размером  $256 \times 256$  пикселей. Такое упорядочивание блоков показано на рис. 7.2. Блоки последовательно нумеруются, начиная с верхнего левого угла. Блок, находящийся в левом верхнем углу — это блок 0. Справа от него находится блок 1 и так далее, по всем строкам, пока не доберемся до блока 63 (нижнего правого блока). В этом случае вы ссылаетесь на блок по его номеру и позволяете процедуре рисования блока самой вычислять его координаты в текстуре.

---

Если вы рисуете второй слева и третий сверху блок, прямоугольник источника будет занимать координаты от 64, 96 до 96, 128. Поскольку размер всех блоков  $32 \times 32$ , необходимо знать только координаты верхнего левого угла блока в текстуре; для вычисления нижней и правой координат блока достаточно просто прибавить 32. Вот как инициализируется структура **RECT** с использованием этих значений:

```
RECT SrcRect;  
SrcRect.left   = 64;           // Левая координата блока  
SrcRect.top    = 96;           // Верхняя координата блока  
SrcRect.right  = SrcRect.left + 32; // Добавляем ширину блока  
SrcRect.bottom = SrcRect.top  + 32; // Добавляем высоту блока
```

Обратите внимание, что если **pSrcRect** присвоить значение **NULL**, то функция рисования будет использовать в качестве блока всю текстуру целиком. Параметр **pScaling** — это вектор, определяющий как вы хотите масштабировать размеры блока. Если вы не будете использовать масштабирование, укажите **NULL**; в ином случае сконструируйте вектор, содержащий коэффициенты масштабирования.

**СОВЕТ**

Возможность масштабировать блоки в Direct3D открывает некоторые удивительные возможности. Вы можете использовать небольшие изображения, скажем  $16 \times 16$  пикселей, и рисовать их в большем размере, например,  $64 \times 64$  пикселя. При рисовании блока текстура растягивается, чтобы соответствовать масштабированному блоку. Применяя для карт такие масштабируемые блоки вы можете значительно увеличить размеры карт и в то же время сэкономить память, используемую для текстур блоков. Я подробнее расскажу об этой технике в разделе «Работа с большими растрами» этой главы.

---

Масштабирование — это единственная из специальных возможностей работы с блоками, которую я собираюсь использовать. Так что вы можете присвоить **pRotationCenter** значение **NULL**, а **Rotation** — 0,0.

Заслуживает внимания параметр **pTransform**, представляющий собой вектор, который сообщает объекту спрайта в каком месте (в экранных координатах от 0 до ширины или высоты окна) рисовать блок. Обратите внимание, что все блоки рисуются сверху вниз и слева направо, подразумевая, что началу координат соответствует левый верхний угол блока.

Последний аргумент функции **Draw** — это **Color**, являющийся значением типа **D3DCOLOR**, используемым для корректировки выходной текстуры. Обычно вы здесь задаете значение 0xFFFFFFFF, чтобы блоки рисовались точно так же, как они выглядят в текстуре, но, используя макрос **D3DCOLOR\_RGBA** вы можете при рисовании изменять цвета или альфа-значения блоков.

Например, чтобы нарисовать блок с половинным альфа-значением, используйте:

```
D3DCOLOR_RGBA(255,255,255,127); // 127 = половина
```

Как видите, значения берутся из диапазона от 0 (цвет или альфа-значение удаляется) до 255 (цвет или альфа-значение остается неизменным). Точно так же, как мы меняли альфа-значение, можно полностью вырезать красную составляющую, используя следующий код:

```
D3DCOLOR_RGBA(0,255,255,255); // 0 = нет цвета
```

Это открывает ряд роскошных возможностей, таких как создание дневных и ночных сцен или блоки, через которые видна расположенная за ними графика (например, окна).

Вернемся к основной теме и посмотрим, как можно использовать функцию **Draw** для рисования блоков. Ниже приведен пример функции, которая получает текстуру, координаты источника и координаты места назначения для рисования блока (обратите внимание, что у координат тип **float**, хотя они определены в экранном пространстве, то есть в размерах области отображения).

```
// Убедитесь, что перед вызовом этой функции вы вызвали
// IDirect3DDevice::BeginScene и загрузили текстуру,
// используемую для хранения блоков
// pSprite = ранее инициализированный объект ID3DXSprite
void DrawTile(float SrcX, float SrcY,
              float DestX, float DestY,
              float TileWidth, float TileHeight,
              float ScaleX, float ScaleY,
              IDirect3DTexture9 *pTileTexture,
              D3DCOLOR Color)
{
    RECT SrcRect; // Прямоугольник источника
```



```
// Задаем прямоугольник источника
SrcRect.left   = SrcX;
SrcRect.top    = SrcY;
SrcRect.right  = SrcRect.left + TileWidth;
SrcRect.bottom = SrcRect.top  + TileHeight;

// Рисуем блок, используя заданные координаты,
// цвет и масштаб. Если вы хотите, чтобы блок
// был нарисован в исходном масштабе, установите
// для ScaleX и ScaleY значения 1.0
pSprite->Draw(pTileTexture, &SrcRect,
              &D3DXVECTOR2(ScaleX, ScaleY), NULL, 0.0f,
              &D3DXVECTOR2(DestX, DestY), Color);
}
```

---

<b>ПРИМЕЧАНИЕ</b>	Как видите, я добавил в функцию <b>DrawTile</b> возможность масштабирования блоков. Это мощная техника, которую я использую дальше в этой главе в разделе «Работа с большими растрами».
-------------------	---

---

Хотя использование объекта **ID3DXSprite** для рисования блоков сперва может показаться странным (и слегка неоптимизированным), я уверяю вас, что он хорошо справляется со своей работой. Далее у нас есть один путь — построить специальный класс, который будет работать с блоками за вас, включая их загрузку и рисование.

## Построение класса для работы с блоками

Поскольку вы узнали, как рисовать блоки, наступило хорошее время для конструирования класса, который будет для вас работать с блоками. Класс должен быть минимальным — достаточно загрузки текстуры блоков и рисования их на указанном устройстве. Взгляните на созданный мной простой класс работы с блоками. Чтобы упростить разработку я интегрировал в него графическое ядро.

```
class cTiles
{
private:
    cGraphics *m_Graphics; // Родительский объект cGraphics

    long m_NumTextures;    // Количество текстур
    cTexture *m_Textures;  // Массив cTexture

    short *m_Widths;       // Массив широт блоков
    short *m_Heights;      // Массив высот блоков
    short *m_Columns;      // Количество столбцов в текстуре

public:
    cTiles();
    ~cTiles();

    // Функции для создания и освобождения интерфейса блока
    BOOL Create(cGraphics *Graphics, long NumTextures);
    BOOL Free();

    // Функции для загрузки и освобождения отдельной текстуры
    BOOL Load(long TextureNum, char *Filename,
```

```

        short TileWidth = 0, short TileHeight = 0,
        D3DCOLOR Transparent = 0,
        D3DFORMAT Format = D3DFMT_A1R5G5B5);
    BOOL Free(long TextureNum=-1);

    // Функции для получения размеров блока
    // и количества блоков в текстуре
    long GetWidth(long TextureNum);
    long GetHeight(long TextureNum);
    long GetNum(long TextureNum);

    // Разрешение и запрещение прозрачности
    BOOL SetTransparent(BOOL Enabled = TRUE);

    // Рисование блока в указанном месте
    BOOL Draw(long TextureNum, long TileNum,
        long ScreenX, long ScreenY,
        D3DCOLOR Color = 0xFFFFFFFF,
        float XScale = 1.0f, float YScale = 1.0f);
};

```

---

**ПРИМЕЧАНИЕ** Все открытые функции класса **cTiles** возвращают значение типа **BOOL**; **TRUE** означает успешное завершение, а **FALSE** — возникновение непредвиденной ошибки.

---

Представленный здесь класс **cTiles** работает выделяя массив объектов **cTexture**, в которых будет храниться блочная графика. Код класса **cTiles** находится на прилагаемом к книге компакт-диске в папке с примерами к данной главе (загляните в папку \BookCode\Chap07\). В следующих разделах приведен код открытых функций, описание того, что они делают, и того, как их вызывать.

### **cTiles::Create**

```

BOOL cTiles::Create(
    cGraphics *Graphics, // Ранее инициализированный объект cGraphics
    long NumTextures);   // Количество создаваемых объектов текстур

```

Первая функция, **cTiles::Create**, выделяет память для массива объектов **cTexture**, в которых будет храниться блочная графика. Убедитесь, что передаете функции **Create** ранее инициализированный объект **cGraphics** и указываете количество текстур, достаточное для хранения блоков.

```

BOOL cTiles::Create(cGraphics *Graphics, long NumTextures)
{
    Free(); // Освобождаем все от существующих данных

    // Проверка ошибок
    if((m_Graphics = Graphics) == NULL)
        return FALSE;
    if((m_NumTextures = NumTextures) == NULL)
        return FALSE;

    // Выделяем память для объектов текстур
    if((m_Textures = new cTexture[m_NumTextures]) == NULL)
        return FALSE;
}

```

```
// Выделяем память для высот, широт и количества столбцов
m_Widths = new long[m_NumTextures];
m_Heights = new long[m_NumTextures];
m_Columns = new long[m_NumTextures];

return TRUE; // Успешное завершение!
}
```

### ***cTiles::Free***

```
BOOL cTiles::Free();
```

Функция не получает параметров, поскольку освобождает все ресурсы и объекты класса. Никакие вызовы **Load**, **Draw** или **Free** не будут работать, пока экземпляр класса **cTiles** не будет заново инициализирован вызовом **cTiles::Create**.

```
BOOL cTiles::Free()
{
    m_Graphics = NULL;

    // Освобождаем все текстуры
    if(m_NumTextures) {
        for(short i = 0; i < m_NumTextures; i++)
            m_Textures[i].Free();
    }
    delete [] m_Textures;
    m_Textures = NULL;

    // Освобождаем массивы высот широт и столбцов
    delete [] m_Widths;
    delete [] m_Heights;
    delete [] m_Columns;
    m_Widths = m_Heights = m_Columns = NULL;
    m_NumTextures = 0;

    return TRUE;
}
```

### ***cTiles::Load***

```
BOOL cTilesLoad(
    long      TextureNum,    // Номер текстуры, куда загружается графика
    char      *Filename,    // Имя загружаемого файла с изображением
                                // (*.bmp)
    short     TileWidth,    // Ширина блоков в изображении
    short     TileHeight,   // Высота блоков в изображении
    D3DCOLOR  Transparent,  // Прозрачный цвет (установите альфа=255)
    D3DFORMAT Format);      // Формат хранения
```

Функция **cTiles::Load** выполняет загрузку текстуры в указанный элемент массива текстур. Например, если вы создали объект **cTiles** для использования пяти текстур, можно указать любой элемент от 0 до 4, в который и будет загружена текстура. На все текстуры ссылаются по их индексу в массиве текстур.

Загружая файл текстуры вы должны указать размер хранящихся в текстуре блоков (измеряется в пикселях). Эти блоки должны быть упакованы в текстуру слева направо и сверху вниз, причем первый блок начинается с верхнего левого пикселя текстуры. Например, у вас может быть текстура, содержащая 64 блока, каждый размером  $32 \times 32$  пикселя. Это означает, что в текстуре 8 строк и 8 столбцов блоков, как показано на рис. 7.2.

Последние два параметра полезны только если вы используете копирование с учетом прозрачности. Присвойте параметру **Transparent** допустимое значение **D3DCOLOR** (используйте **D3DCOLOR\_RGBA** или аналогичный макрос, убедившись, что указали альфа-значение 255) и либо оставьте для **Format** значение по умолчанию **D3DFMT\_A1R5G5B5**, либо задайте собственный формат из предоставляемого Direct3D списка допустимых форматов.

Вот код функции **Load**:

```

BOOL cTiles::Load(long TextureNum, char *Filename,
                  short TileWidth, short TileHeight,
                  D3DCOLOR Transparent, D3DFORMAT Format)
{
    // Проверка ошибок
    if(TextureNum >= m_NumTextures || m_Textures == NULL)
        return FALSE;

    // Освобождаем требуемую текстуру
    Free(TextureNum);

    // Загружаем текстуру
    if(m_Textures[TextureNum].Load(m_Graphics, Filename,
                                   Transparent, Format) == FALSE)
        return FALSE;

    // Сохраняем значение ширины (получаем ширину текстуры,
    // если не задан параметр TileWidth)
    if(!TileWidth)
        m_Widths[TextureNum] = m_Textures[TextureNum].GetWidth();
    else
        m_Widths[TextureNum] = TileWidth;

    // Сохраняем значение высоты (получаем высоту текстуры,
    // если не задан параметр TileHeight)
    if(!TileHeight)
        m_Heights[TextureNum] = m_Textures[TextureNum].GetHeight();
    else
        m_Heights[TextureNum] = TileHeight;

    // Вычисляем сколько столбцов блоков находится в текстуре.
    // Это используется для ускорения вычислений при рисовании блоков
    m_Columns[TextureNum] = m_Textures[TextureNum].GetWidth() /
        m_Widths[TextureNum];

    return TRUE;
}

```

***cTiles::Free***

```
BOOL cTiles::Free(long TextureNum); // Номер освобождаемой текстуры
```

Функция освобождает отдельную текстуру из массива, оставляя возможность ее повторного использования путем вызова **cTiles::Load**. Просто укажите номер освобождаемой текстуры в аргументе **TextureNum**. Функцию можно использовать для освобождения старых текстур и выделения места для новых.

```
BOOL cTiles::Free(long TextureNum)
{
    // Проверка ошибок
    if(TextureNum >= m_NumTextures || m_Textures == NULL)
        return FALSE;

    // Освобождаем ресурсы отдельной текстуры
    m_Textures[TextureNum].Free();

    return TRUE;
}
```

***cTiles::GetWidth, cTiles::GetHeight u cTiles::GetNum***

```
long cTiles::GetWidth(long TextureNum); // Номер интересующей текстуры
long cTiles::GetHeight(long TextureNum);
long cTiles::GetNum(long TextureNum);
```

Вы используете эти три функции для получения ширины, высоты и количества блоков в текстуре, соответственно. Вы будете редко обращаться к этим функциям напрямую, но их полезно иметь в классе работы с блоками.

```
long cTiles::GetWidth(long TextureNum)
{
    // Проверка ошибок
    if(TextureNum >= m_NumTextures || m_Widths == NULL)
        return 0;

    return m_Widths[TextureNum];
}

long cTiles::GetHeight(long TextureNum)
{
    // Проверка ошибок
    if(TextureNum >= m_NumTextures || m_Heights == NULL)
        return 0;

    return m_Heights[TextureNum];
}

long cTiles::GetNum(long TextureNum)
{
    // Проверка ошибок
    if(TextureNum >= m_NumTextures || m_Textures == NULL ||
        m_Columns == NULL || m_Widths == NULL || m_Heights == NULL)
        return 0;
}
```

```

        return m_Columns[TextureNum] +
               m_Textures[TextureNum].GetHeight() /
               m_Heights[TextureNum];
    }

```

### ***cTiles::SetTransparent***

```

BOOL cTiles::SetTransparent(BOOL Enabled); // Разрешить/запретить

```

Функция **cTiles::SetTransparent** разрешает или запрещает альфа-проверку, а это значит, что текстуры, загруженные с соответствующим цветовым ключом и форматом когда разрешено, будут использовать копирование с учетом прозрачности. По умолчанию параметру **Enable** присваивается **TRUE**. Взгляните на код функции **SetTransparent**:

```

BOOL cTiles::SetTransparent(BOOL Enabled)
{
    // Проверка ошибок
    if(m_Graphics == NULL)
        return FALSE;

    return m_Graphics->EnableAlphaTesting(Enabled);
}

```

### ***cTiles::Draw***

```

BOOL cTiles::Draw(
    long      TextureNum, // Номер рисуемой текстуры
    long      TileNum,    // Номер рисуемого блока
    long      ScreenX,    // Координата X
    long      ScreenY,    // Координата Y
    D3DCOLOR Color,       // Значение модулирования цвета
    float     XScale,     // Коэффициент масштабирования по x
    float     YScale);    // Коэффициент масштабирования по y

```

Эта функция используется для рисования ваших блоков (через метод **Blit** вашего объекта **ID3DXSprite**). Как только текстура загружена в массив, вы можете рисовать отдельные, содержащиеся в ней блоки. Все блоки нумеруются, начиная с нуля; нумерация идет с верхнего левого угла и увеличивается слева направо и сверху вниз. Например, в текстуре с 64 блоками (в сетке  $8 \times 8$ ) нумерация идет от 0 до 63. Блок 0 — это верхний левый блок, под ним располагается блок 8, а справа от него — блок 1.

При рисовании вам надо указать экранные координаты в переменных типа **long** (позднее они будут преобразованы в значения **float**), а также коэффициенты масштабирования (для увеличения или уменьшения размера блока относительно исходного). Как упоминалось ранее, значение модуляции применяется для изменения цвета или альфа-значения с использованием интерфейса **ID3DXSprite**.

```

BOOL cTiles::Draw(long TextureNum, long TileNum,
                  long ScreenX, long ScreenY,
                  D3DCOLOR Color, float XScale, float YScale)
{

```

```
    long SrcX, SrcY;

    // Проверка ошибок
    if(m_Graphics == NULL)
        return FALSE;
    if(TextureNum >= m_NumTextures || m_Textures == NULL)
        return FALSE;

    // Вычисление координат источника блока в текстуре
    SrcX = (TileNum % m_Columns[TextureNum]) * m_Widths[TextureNum];
    SrcY = (TileNum / m_Columns[TextureNum]) * m_Heights[TextureNum];

    return m_Textures[TextureNum].Blit(ScreenX, ScreenY,
                                       SrcX, SrcY,
                                       m_Widths[TextureNum],
                                       m_Heights[TextureNum],
                                       XScale, YScale);
}
```

## Использование класса работы с блоками

Вот пример, который загружает две текстуры с блоками. Первая текстура содержит 64 блока, каждый размером  $32 \times 32$  пикселя. Вторая текстура содержит 16 блоков, каждый размером  $64 \times 64$  пикселя.

```
// Graphics = ранее инициализированный объект cGraphics
cTiles Tile;

// Создаем класс блоков с местом для двух текстур
Tile.Create(Graphics, 2);

// Загружаем обе текстуры, указывая,
// что черный цвет будет прозрачным
Tile.Load(0, "Tiles1.bmp", 32, 32,
          D3DCOLOR_RGBA(0,0,0,255), D3DFMT_A1R5G5B5);
Tile.Load(1, "Tiles2.bmp", 64, 64,
          D3DCOLOR_RGBA(0,0,0,255), D3DFMT_A8R8G8B8);

// Рисуем пару блоков из первой текстуры без прозрачности
Tile.SetTransparent(FALSE);

// Блок 0 (в точке 128, 128) и 3 (в точке 0, 0)
Tile.Draw(0, 0, 128, 128);
Tile.Draw(0, 3, 0, 0);

// Рисуем пару блоков из второй текстуры с прозрачностью
Tile.SetTransparent(TRUE);

// Блок 1 (в точке 28, 18) и 16 (в точке 100, 90)
Tile.Draw(1, 1, 28, 18);
Tile.Draw(1, 16, 100, 90);

// Освобождаем класс работы с блоками и текстуры
Tile.Free();
```

Вот и все. Класс работы с блоками компактен и замечательно дополнит вашу библиотеку двумерной графики, поскольку выполняет за вас все операции, связанные с рисованием блоков. От вас требуется только предоставить классу графику, которую вы хотите использовать, и все готово к работе.

## Основы работы с блочной графикой

Время пришло. Вы готовы создать движок блочной графики (не библиотеку рисования блоков). Хотя этот механизм блочной графики почти так же стар, как компьютерные игры, он все еще остается наиболее используемой техникой двумерной графики. Фактически, почти все игры, созданные для портативной игровой консоли Nintendo Gameboy Advance используют представленные ниже технологии блочной графики. Эта игровая система использует исключительно блочную графику в той или иной форме, чтобы предоставить вам совершенную графику на портативном игровом устройстве (я ведь уже говорил это?).

Теперь у вас есть шанс повторить несколько старых техник, которые могут помочь в ваших проектах.

### Рисование основной карты

Рисование основной блочной карты — это быстрый и безболезненный процесс; вам необходимо в цикле перебирать столбцы и строки, рисуя те блоки которые вы проходите. Общее количество рисуемых блоков зависит от их размера и разрешения экрана. Например, если размеры игровой области на экране  $384 \times 384$  пикселя, и используются блоки размером  $64 \times 64$  пикселя, то у вас на экране поместится 6 рядов по 6 блоков в каждом, то есть всего 36 блоков.

Сейчас вернемся к рис. 7.1, где изображены четыре блока, карта и результат визуализации карты. Чтобы представить карту в виде массива, используя номера блоков, указываемые при их рисовании, сделайте следующее:

```
char Map[6][6] = { // Map[y][x]
    { 3, 3, 3, 3, 1, 1 },
    { 3, 3, 3, 3, 3, 3 },
    { 2, 3, 3, 2, 3, 3 },
    { 3, 3, 3, 3, 3, 3 },
    { 3, 3, 3, 3, 3, 1 },
    { 4, 4, 1, 1, 1, 1 }
};
```

Чтобы нарисовать показанную выше карту, вы должны перебрать все элементы массива и для каждого нарисовать соответствующий блок:

```
// Tile = ранее инициализированный и загруженный объект cTiles
for(short row = 0; row < 6; row++) {
    for(short column = 0; column < 6; column++) {

        // Получаем номер рисуемого блока
        char TileNum = Map[row][column];

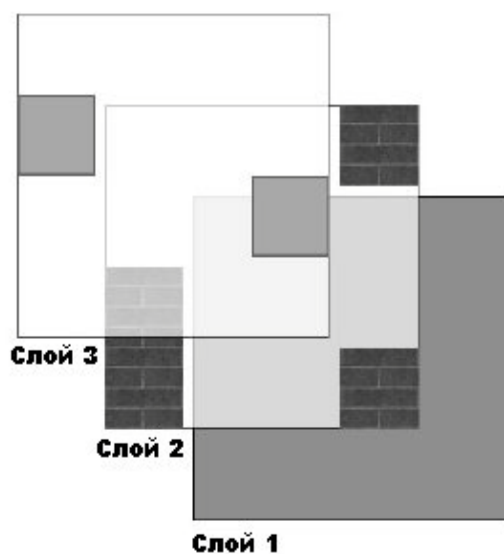
        // Рисуем блок (размером 64 x 64), соответствующий
        // TileNum из первой загруженной текстуры
        Tile.Draw(0, TileNum, column*64, row*64);
    }
}
```



Если у вас есть набор из трех блоков и показанная выше карта (и функция рисования), вы получите на экране визуализированную карту, аналогичную той, что была показана раньше на рис. 7.1.

## Использование нескольких слоев

Приложив немного усилий вы можете усовершенствовать ваш механизм работы с блочными картами. В большинстве использующих блочную графику игр применяется несколько слоев (сцены помещаются одна над другой, как показано на рис. 7.3) для создания некоторых замечательных эффектов. Например, нарисовав сначала землю, затем персонажей, а затем слой с другими перекрывающим объектами, вы получите имитацию трехмерной сцены.



*Рис. 7.3. Чтобы создать имитацию трехмерного окружения, вы располагаете слои карты один над другим*

Чтобы использовать несколько слоев вы просто объявляете еще один массив карты (один массив для каждого слоя) и заполняете его информацией о блоках. Начните с первого слоя и нарисуйте все содержащиеся в нем блоки. Когда вы нарисуете последний блок слоя, перейдите к следующему слою и рисуйте его блоки. Продолжайте, пока не нарисуете все слои.

Предположим, что вы используете четыре слоя. У вас есть базовый слой для земли и объектов, которые не могут скрывать персонажей, слой для рисования персонажей, слой для тех объектов, которые могут скрывать персонажей и слой, который скрывает все (например, для плывущих в вышине облаков).

Теперь вы создаете пять массивов карт и заполняете каждый массив данными о блоках, которые будут рисоваться. Войдя в функцию визуализации карты мы проходим по каждому массиву слоя карты и рисуем его на экране. Это продолжается пока не нарисованы все слои. Код для рисования многослойной карты может выглядеть так:

```
// Tile = ранее инициализированный и загруженный объект cTiles
char Map[5][10][10]; // Данные карты, подразумеваем,
                      // что они уже загружены
```

```
// Перебираем в цикле все слои
for(short Layer = 0; Layer < 5; Layer++) {

    // Перебираем строки и столбцы слоя
    for(short row = 0; row < 10; row++) {
        for(short column = 0; column < 10; column++) {

            // Получаем номер рисуемого блока
            char TileNum = Map[Layer][row][column];

            // Рисуем блок (размером 32 x 32)
            // соответствующий TileNum из первой
            // загруженной текстуры
            Tile.Draw(0, TileNum, column*32, row*32);
        }
    }
}
```

## Добавление объектов

Недавно я упомянул, что персонажи рисуются как слой карты; такое решение не совсем верно, поскольку персонажи могут свободно перемещаться по всему миру и не соответствуют лежащей в основе многослойных карт теории. Вместо этого персонажи и другие перемещаемые объекты нужно рисовать как свободно позиционируемые блоки; для них не требуется использовать массив карты. Если говорить более подробно, вы отслеживаете все персонажи и объекты по их относительным координатам в мире; затем, когда они появляются в поле зрения, вы преобразуете эти относительные координаты в координаты, пригодные для рисования на экране.

Чтобы сейчас не усложнять вещи, давайте создадим структуру, хранящую координаты объекта и единственный номер рисуемого блока (который представляет объект):

```
typedef struct sObject {
    long XPos, YPos; // Координаты объекта
    char Tile;       // Рисуемый блок
} sObject;
```

Обратите внимание, что сейчас я рассматриваю все, что может свободно перемещаться, включая персонаж игрока. Следя за этими персонажами, вы должны гарантировать, что они будут добавлены к списку рисуемых объектов. Этот список объектов является простым массивом, объявление которого выглядит так:

```
#define MAX_OBJECTS 1024
sObject Objects[MAX_OBJECTS]; // Список для 1024 объектов
```

Для каждого кадра отслеживайте количество находящихся в нем объектов, которые будут нарисованы, используя переменную:

```
long NumObjectsToDraw = 0;
```

В каждом кадре вашей игры **NumObjectToDraw** сбрасывается в 0, а затем, когда объект добавляется к списку, счетчик увеличивается. Например, для добавления объекта вы можете использовать следующую функцию:

```
void AddObject(long XPos, long YPos, char Tile)
{
    if (NumObjectsToDraw < MAX_OBJECTS) {
        Objects[NumObjectsToDraw].XPos = XPos;
        Objects[NumObjectsToDraw].YPos = XPos;
        Objects[NumObjectsToDraw].Tile = Tile;

        NumObjectsToDraw++;
    }
}
```

Когда придет время визуализировать объекты, вы просто сканируете заданное число визуализируемых объектов списка и рисуете соответствующие им блоки. Обратите внимание, что координаты карты, используемые при визуализации карты на дисплее должны смещать каждый объект:

```
// Tiles = ранее инициализированный и загруженный объект cTiles
// MapXPos, MapYPos = координаты карты
for(i = 0; i < NumObjectsToDraw; i++)
    Tiles.Draw(0, Objects[i].Tile,
               Objects[i].XPos - MapXPos * TileWidth,
               Objects[i].YPos - MapYPos * TileHeight);
```

---

**ПРИМЕЧАНИЕ**

Вы обратили внимание, что во фрагменте кода рисования я умножал координаты карты на ширину и высоту блока (определенные как переменные **TileWidth** и **TileHeight**). Вы спрашиваете зачем? Да потому что объекты не привязаны к сетчатой структуре карты! Объекты используют *точные координаты (fine coordinate)*, о которых вы прочтете в последующих разделах.

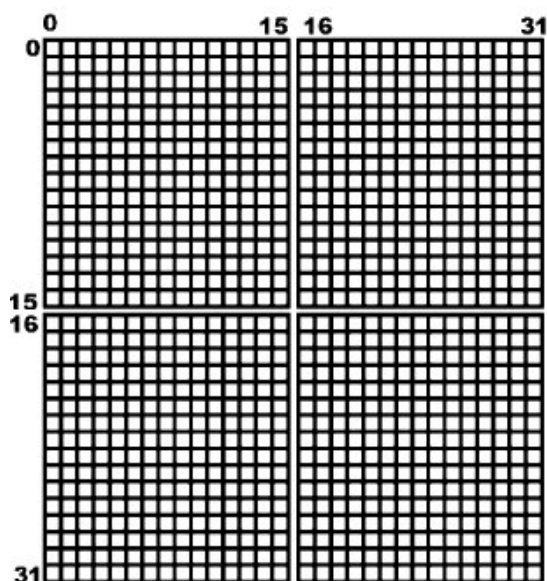
---

## Плавная прокрутка

Поэкспериментировав с блочными картами вы обнаружите, что для больших карт необходима возможность прокрутки, чтобы игрок мог осматривать всю карту. В текущем воплощении механизм работы с блоками производит похожий на рывок сдвиг всякий раз, когда вы меняете координаты рисования карты. Чтобы улучшить визуальное впечатление надо сделать это движение более плавным, используя технику, известную как *плавная прокрутка (smooth scrolling)*.

Чтобы визуально представить плавную прокрутку, вообразите блочную карту в виде большого растрового изображения. У каждого пикселя растра есть собственная пара координат, называемая *точные координаты (fine coordinates)* карты. У каждой группы пикселей, представляющей блок, есть собственный набор координат на карте. Например, если размер блока  $16 \times 16$  пикселей, а карта представляет собой массив  $10 \times 10$ , размеры визуализированного изображения карты будут  $160 \times 160$  пикселей (значит у

карты разрешение точных координат  $160 \times 160$  пикселей). Такой пример показан на рис. 7.4.



*Рис. 7.4. Карта, размером  $2 \times 2$  блока использует блоки размером  $16 \times 16$  пикселей, так что разрешение точных координат будет  $32 \times 32$ . Обратите внимание, что у каждого пикселя есть собственная пара координат*

Если игровое поле занимает на экране область только  $100 \times 100$  пикселей, будет рисоваться только часть карты. Конечно, эти пиксели относятся к блокам, которые должны рисоваться в соответствующем образом выровненных местах экрана. Для этого вы должны задавать координаты карты на уровне пикселей (используя точные координаты) и уметь рисовать только небольшие фрагменты блоков.

Я не буду переусложнять предмет обсуждения; для использования плавной прокрутки достаточно добавить несколько строчек кода. Когда вы рисуете блоки карты надо всего лишь слегка сместить их, чтобы выровнять по точным координатам карты.

Чтобы вычислить смещение при рисовании блоков, вы просто берете точные координаты карты, с которых хотите начать рисование (точные координаты карты относятся к первому рисуемому пикселю, находящемуся в верхнем левом углу экрана) и вычисляете несколько переменных, как показано ниже:

```
// Предполагаем, что FineX и FineY - это
// используемые точные координаты
// TileWidth и TileHeight - это размеры блока
// Вычисляем действительные координаты блока карты
long MapX, MapY; // Координаты в массиве карты

MapX = FineX / TileWidth; // Получаем координату карты X
MapY = FineY / TileHeight; // Получаем координату карты Y

// Вычисляем смещение блока в пикселях
long XOff, YOff; // Смещение в пикселях
XOff = FineX % TileWidth; // Смещение по X
YOff = FineY % TileHeight; // Смещение по Y
```

Обратите внимание, что я вычисляю четыре переменных. Первую пару координат, **MapX** и **MapY**, вы используете когда обращаетесь к массиву

карты. Например, если вы указываете точные координаты карты 8, 8 (и размер блока  $16 \times 16$ ), действительные координаты в массиве карты будут 0, 0 (поскольку пиксель с координатами 8, 8 находится внутри верхнего левого блока карты).

Вторая пара координат, **XOff** и **YOff**, являются смещениями рисуемых блоков. Для вычисления **XOff** и **YOff** вы берете модуль (остаток) от деления точных координат на размеры блока. Потом, всякий раз, когда вы хотите нарисовать блок карты (или любой блок, который использует точные координаты карты), вычтите значение **XOff** из координаты X блока, и значение **YOff** из координаты Y блока.

Вот как смещения плавной прокрутки работают в цикле визуализации:

```
// Tile = ранее инициализированный и загруженный объект cTiles
for(short row = 0; row < 11; row++) {
    for(short column = 0; column < 11; column++) {
        // Получаем номер рисуемого блока
        char TileNum = Map[row][column];

        // Рисуем блок (размером 32x32), связанный с TileNum
        // из первой загруженной текстуры
        Tile.Draw(0, TileNum, column*32-XOff, row*32-YOff);
    }
}
```

---

**ПРИМЕЧАНИЕ**

Когда вы добавляете к своей графической библиотеке возможность плавной прокрутки, то в цикле визуализации должны рисовать дополнительную строку и столбец блоков, как в приведенном выше примере. Сцена может содержать карту  $10 \times 10$ , но поскольку ее можно плавно прокрутить в любом направлении, необходимо заполнять левый и нижний края, для чего и рисуется фрагмент  $11 \times 11$  блоков.)

---

## Карта и мышь

Даже если вы очень долго не играли в компьютерные игры, мимо вашего внимания не мог проскользнуть тот факт, что большинство игр активно используют мышь. Управление в стиле «укажи и щелкни» является действительно интуитивно понятным. Игрок просто щелкает по какому-то месту на экране, и его персонаж идет, куда было указано; или можно щелкнуть по предмету, чтобы взять его. А если вы захотите использовать такие возможности в вашей игре — как определить по чему щелкнул игрок?

Для прямоугольных карт у вас есть два способа определить по чему щелкнул игрок. Первый, и простейший, способ заключается в делении координат мыши на размер блока. Это дает вам координаты в массиве карты. Если вы используете плавную прокрутку, необходимо принимать во внимание смещение, используемое при рисовании блоков.

Для примера предположим, что вы используете плавную прокрутку карты с блоками размером  $32 \times 32$  пикселя. Координаты карты (в точных

координатах) 48, 102, а координаты мыши — 45, 80. Сначала вычислим значения смещений для плавной прокрутки:

```
XOff = 48 % 32; // FineX / TileWidth
YOff = 102 % 32; // FineY / TileHeight
```

Затем вычтем значения смещений из координат мыши и прибавим точные координаты карты, с которых она рисуется:

```
// Подразумеваем, что mouseX, mouseY - координаты мыши
long MouseFineX, MouseFineY; // Точные координаты щелчка
MouseFineX = mouseX - XOff + 48;
MouseFineY = mouseY - YOff + 102;
```

Теперь у вас есть точные координаты карты для того пикселя, по которому щелкнул пользователь. Осталось разделить **MouseFineX** и **MouseFineY** на размеры блока, чтобы получить координаты элемента массива:

```
long MouseMapX, MouseMapY; // Координаты щелчка по карте
MouseMapX = MouseFineX / 32;
MouseMapY = MouseFineY / 32;
```

Хотя приведенного выше фрагмента кода в большинстве случаев достаточно, иногда он не подходит, например при работе со свободно перемещающимися объектами, такими как игровые персонажи. Предположим, игрок щелкает по персонажу, который пересекает экран. Поскольку персонажи не относятся к массиву данных карты, вы используете метод сканирования рисуемых блоков и сравниваете их координаты с координатами указателя мыши. Это простой вопрос *проверки границ (bounds checking)*.

Предположим, что размер блока  $64 \times 64$  пикселя, и он рисуется в точке экрана с координатами 48, 100. Если указатель мыши находится в точке экрана с координатами 60, 102, то он попадает в блок, поскольку блок занимает область от 48, 100 до 112, 164. Вы можете написать небольшую функцию, которая получает координаты блока (в экранном пространстве), размеры блока и координаты указателя мыши и возвращает **TRUE**, если указатель находится внутри блока и **FALSE** в ином случае.

```
BOOL IsMouseTouchingTile(
    long TileX, long TileY,           // Координаты блока
    long TileWidth, long TileHeight, // Размеры блока
    long mouseX, long mouseY)        // Координаты мыши
{
    // Проверяем выход за левую сторону
    if(mouseX < TileX) return FALSE;

    // Проверяем выход за правую сторону
    if(mouseX >= TileX + TileWidth) return FALSE;

    // Проверяем выход за верхнюю сторону
    if(mouseY < TileY) return FALSE;

    // Проверяем выход за нижнюю сторону
    if(mouseY >= TileY + TileHeight) return FALSE;
```

```
// Указатель находится внутри блока
return TRUE; // Возвращаем успех
}
```

Возвращаясь к примеру определения того, по какому блоку щелкнул игрок, можно воспользоваться приведенной функцией, чтобы выяснить попадает ли указатель мыши в границы блока:

```
if(IsMouseTouchingTile(48,100,64,64,60,102) == TRUE)

    // Мышь касается блока!

else

    // Мышь не касается блока
```

## Создание класса карты

Поскольку вы уже узнали о том как использовать блоки, слои, плавную прокрутку и визуализировать объекты, давайте завернем все это в небольшой класс (и вспомогательные структуры). Чтобы сохранить плавность выполнения игры надо сперва ограничить количество свободно перемещающихся блоков (спрайтов), которое будет рисоваться в каждом кадре. Определение макроса прекрасно выполняет задачу информирования класса карты о том, сколько спрайтов может быть нарисовано в кадре:

```
#define MAX_OBJECTS 1024
```

Затем следует структура данных спрайта в которой хранятся координаты и номер блока объекта спрайта, который будет нарисован при вызове функции визуализации очередного кадра:

```
typedef struct {
    long XPos, YPos;
    char Tile;
} sObject;
```

Далее идет объявление класса карты. Он содержит массив структур **sObject** и массив слоев карты. Размеры карты хранятся в двух переменных, **m\_Width** и **m\_Heigth**, которые устанавливаются при инициализации класса карты вызовом **Create**.

Каждый экземпляр класса карты может хранить огромное количество слоев (около миллиона). Вы храните данные каждого слоя блоков в отдельном массиве **m\_Map**. Поскольку размеры карты фиксируются при ее создании, вы получаете доступ к данным каждого слоя блоков вычисляя текущее смещение в массиве **m\_Map** и осуществляя чтение или запись этого слоя через указатель. Доступ к сданным слоев демонстрируют функции класса **SetMapData** и **Render**.

Что касается используемой графики, класс карты ограничен одним набором блоков (для хранения изображений блоков используется только одна текстура). Вы информируете класс карты какой объект класса блоков

будет использоваться для рисования карты через функцию **UseTiles**, которую вы увидите в объявлении класса.

Поговорив об объявлении класса карты давайте посмотрим на него:

```
class cMap
{
    private:
        long m_Width, m_Height;           // Ширина и высота карты
        long m_NumLayers;                 // Количество слоев
        char *m_Map;                      // Массив для данных блоков

        cTiles *m_Tiles;                  // Класс cTile для работы с блоками

        long m_NumObjectsToDraw;          // Количество рисуемых объектов
        sObject m_Objects[MAX_OBJECTS];   // Список объектов

    public:
        cMap(); // Конструктор
        ~cMap(); // Деструктор

        // Функции создания и освобождения класса карты
        BOOL Create(long NumLayers, long Width, long Height);
        BOOL Free();

        // Функция для установки данных слоя карты
        BOOL SetMapData(long Layer, char *Data);

        // Функции очистки и добавления объекта в список
        void ClearObjectList();
        BOOL AddObject(long XPos, long YPos, char Tile);

        char *GetPtr(long Layer); // Получение указателя на массив карты
        long GetWidth(); // Получение ширины карты
        long GetHeight(); // Получение высоты карты

        // Назначаем объект класса cTile, который будет
        // использоваться для рисования блоков карты
        BOOL UseTiles(cTiles *Tiles);

        // Визуализируем карту, используя заданные координаты
        // верхнего левого угла карты, количество рисуемых строк
        // и столбцов и слой, используемый для рисования объектов
        BOOL Render(long XPos, long YPos,
                    long NumRows, long NumColumns,
                    long ObjectLayer);
};
```

Большинство функций прокомментированы и самодокументируемы, так что давайте пойдем дальше и посмотрим на полный код класса **cMap**. Код начинается с конструктора и деструктора, обеспечивающих установку данных класса в известное состояние и освобождение всех ресурсов при уничтожении объекта класса:

```
cMap::cMap()
{
    m_Map = NULL;
    m_Tiles = NULL;
    m_NumObjectsToDraw = 0;
    m_Width = m_Height = 0;
}
```



```
cMap::~cMap()
{
    Free();
}
```

Чтобы использовать класс карты вы должны сначала объявить экземпляр класса и вызвать функцию **Create**. Функция **Create** получает количество слоев в карте, а также ширину и высоту карты:

```
BOOL cMap::Create(long NumLayers, long Width, long Height)
{
    // Освобождаем предыдущую карту
    Free();

    // Сохраняем количество слоев, ширину и высоту
    m_NumLayers = NumLayers;
    m_Width      = Width;
    m_Height     = Height;

    // Выделяем память для данных карты
    if((m_Map = new char[m_NumLayers*m_Width*m_Height]) == NULL)
        return FALSE;

    // Очищаем карту
    ZeroMemory(m_Map, m_NumLayers*m_Width*m_Height);

    // Сбрасываем количество рисуемых объектов
    m_NumObjectsToDraw = 0;

    return TRUE;
}
```

Говоря кратко и по существу, функция **Create** выделяет память для массива значений **char**, в котором будет храниться информация. Размер массива определяется шириной, высотой и количеством используемых столбцов; перемножьте эти три значения и вы получите итоговый размер массива.

Чтобы обеспечить освобождение памяти, занимаемой массивом слоев карты, когда работа с классом карты завершена, вы вызываете функцию **Free**:

```
BOOL cMap::Free()
{
    // Освобождаем массив карты
    delete [] m_Map;
    m_Map = NULL;
    m_Width = m_Height = 0;
    m_NumLayers = 0;

    return TRUE;
}
```

Чтобы заполнить массив слоев карты полезной информацией о блоках вы сперва создаете представляющий слой массив значений **char**, как было показано раньше в разделе «Рисование основной карты». Используя этот массив в качестве аргумента вы обращаетесь к функции **SetMapData**,

указав также номер слоя карты, в который вы хотите скопировать данные из массива:

```

BOOL cMap::SetMapData(long Layer, char *Data)
{
    // Проверка ошибок
    if(Layer >= m_NumLayers)
        return FALSE;

    // Копирование данных
    memcpy(&m_Map[Layer*m_Width*m_Height], Data, m_Width*m_Height);

    return TRUE;
}

```

Для каждого визуализируемого кадра (или как минимум, каждый раз когда вы рисуете карту), вы конструируете список объектов спрайтов, которые должны быть нарисованы. Сперва вызовом **ClearObjectList** вы подготавливаете класс карты к началу приема информации о спрайтах, которая будет использоваться при визуализации:

```

void cMap::ClearObjectList()
{
    m_NumObjectsToDraw = 0;
}

```

Чтобы добавить спрайт к списку рисуемых в кадре, вызовите **AddObject**, предоставив экранные координаты, где будет нарисован указанный блок:

```

BOOL cMap::AddObject(long XPos, long YPos, char Tile)
{
    if(m_NumObjectsToDraw < MAX_OBJECTS) {
        m_Objects[m_NumObjectsToDraw].XPos = XPos;
        m_Objects[m_NumObjectsToDraw].YPos = YPos;
        m_Objects[m_NumObjectsToDraw].Tile = Tile;
        m_NumObjectsToDraw++;

        return TRUE;
    }
    return FALSE;
}

```

Если вы хотите напрямую менять данные слоев карты, можете вызвать методы **GetPtr**, **GetWidth** и **GetHeight** класса карты, возвращающие указатель на массив данных слоев карты, ширину карты и высоту карты, соответственно:

```

char *cMap::GetPtr(long Layer)
{
    if(Layer >= m_NumLayers)
        return NULL;

    return &m_Map[Layer*m_Width*m_Height];
}

```

```
long cMap::GetWidth()
{
    return m_Width;
}

long cMap::GetHeight()
{
    return m_Height;
}
```

Раньше я говорил, что для рисования карты вы можете использовать только один набор блоков. Чтобы сообщить классу карты, какой объект класса блоков использовать, передайте указатель на экземпляр класса в функцию класса карты **UseTiles**:

```
BOOL cMap::UseTiles(cTiles *Tiles)
{
    if((m_Tiles = Tiles) == NULL)
        return FALSE;

    return TRUE;
}
```

Наконец вы добрались до функции **Render**, которая визуализирует карту на экране, используя предоставленные вами точные координаты карты, а также количество рисуемых строк и столбцов. Если вы используете спрайты, то должны также указать, после какого слоя их рисовать, установив аргумент **ObjectLayer**:

```
BOOL cMap::Render(long XPos, long YPos,
                  long NumRows, long NumColumns,
                  long ObjectLayer)
{
    long MapX, MapY;
    long XOff, YOff;
    long Layer, Row, Column, i;

    char TileNum;
    char *MapPtr;

    // Проверка ошибок
    if(m_Map == NULL || m_Tiles == NULL)
        return FALSE;

    // Вычисление переменных для плавной прокрутки
    MapX = XPos / m_Tiles->GetWidth(0);
    MapY = YPos / m_Tiles->GetHeight(0);
    XOff = XPos % m_Tiles->GetWidth(0);
    YOff = YPos % m_Tiles->GetHeight(0);

    // Цикл перебора слоев
    for(Layer = 0; Layer < m_NumLayers; Layer++) {

        // Получаем указатель на данные карты
        MapPtr = &m_Map[Layer*m_Width*m_Height];

        // Цикл по строкам и столбцам
        for(Row = 0; Row < NumRows+1; Row++) {
            for(Column = 0; Column < NumColumns+1; Column++) {
```

```

// Получаем номер рисуемого блока (и рисуем его)
TileNum = MapPtr[(Row+MapY) * m_Width+Column+MapX];
m_Tiles->Draw(0, TileNum,
              Column * m_Tiles->GetWidth(0) - XOff,
              Row * m_Tiles->GetHeight(0) - YOff);
    }
}

// Если это слой объектов - рисуем объекты
if(Layer == ObjectLayer) {
    for(i = 0; i < m_NumObjectsToDraw; i++)
        m_Tiles->Draw(0, m_Objects[i].Tile,
                      m_Objects[i].XPos - XOff,
                      m_Objects[i].YPos - YOff);
    }
}
return TRUE;
}

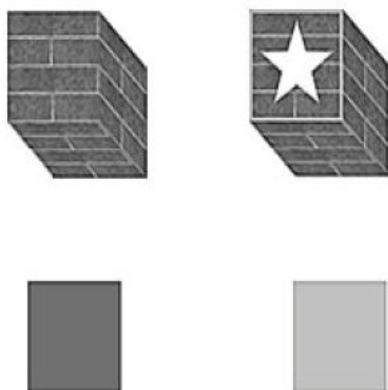
```

Каждая функция в **cMap** прямолинейна и, по сути, повторяет информацию, изложенную ранее в разделе «Основы использования блочной графики». Я предлагаю вам посмотреть примеры программ к этой главе, чтобы увидеть класс **cMap** в действии. Вы найдете их на прилагаемом к книге CD-ROM (загляните в папку BookCode\Chap07\).

## Псевдотрехмерные блоки

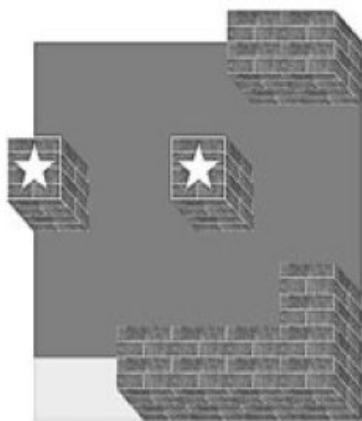
Почти не меняя вашу основную библиотеку блочной графики вы можете начать использовать псевдотрехмерные блоки, что добавит вашей графике дополнительное визуальное измерение. Ваша графика будет показывать ширину, высоту и глубину. Единственное изменение — графика блоков.

Вернемся к предыдущему примеру блочной графики, заменим несколько блоков и посмотрим что получилось. Взгляните на рис. 7.5, чтобы увидеть новый набор используемых блоков.



**Рис. 7.5.** Группа псевдотрехмерных блоков добавляет скучным, плоским картам дополнительное измерение. Псевдотрехмерные блоки демонстрируют ширину, высоту и глубину

Теперь, используя ту же карту, что и в предыдущем примере, снова запустите функцию рисования карты (но на этот раз с новыми блоками). В результате вы должны получить сцену, похожую на рис. 7.6.



**Рис. 7.6.** Используя псевдотрехмерные блоки и карту вы получаете фантастические результаты!

Те же самые техники, которые использовались для обычных блоков (такие, как определение места щелчка мыши по карте) применимы почти без изменений и для псевдотрехмерных блоков — надо только учесть изменившиеся размеры блоков.

## Работа с большими растрами

Последняя техника двумерной графики, которую я собираюсь показать вам, обычно называется *работа с большими растрами* (*big bitmap engine*). Используемая в популярных играх, таких как Baldur's Gate, техника работы с большими растрами позволяет по-новому взглянуть на сферу применения двумерной графики. В названии все сказано — большие растры. Техника использует в качестве уровней огромные растровые изображения, а значит визуальное качество карты значительно улучшается по сравнению с использованием маленьких блоков.

Из-за особенностей памяти текстур используемые для хранения уровней растры делятся на несколько файлов с изображениями. Например, размер уровня может быть  $1024 \times 1024$  пикселя. Поскольку вас ограничивают предельные размеры текстур в Direct3D, вы должны разделить этот уровень на 16 текстур, размером  $256 \times 256$  пикселей (упорядоченных в сетку  $4 \times 4$ ). Фактически вы разделяете уровень на 16 блоков.

Поскольку для визуализации уровня вы используете блоки, перед вами открываются некоторые замечательные возможности. Например, вы можете экономить память, повторно используя блоки уровня. Возьмем, к примеру, траву; это общее изображение, которое может применяться несколько раз, и на большой картине трава будет прекрасно гармонировать с остальным изображением.

Вы можете не только повторно использовать блоки, но и масштабировать их для создания еще больших уровней. Например, увеличив масштаб блоков в два раза вы в четыре раза увеличите размер уровня. А как насчет увеличения уровня в восемь или шестнадцать раз? Это все возможно, и используя несколько слоев блоков вы можете конструировать исключительно большие (и прекрасные) уровни.

**ПРИМЕЧАНИЕ**

Каждый слой карты в вашем движке может иметь собственный набор блоков, которые могут произвольно масштабироваться. Например, самый нижний слой может использовать огромные блоки размером  $256 \times 256$  пикселей, масштабируемые вдвое относительно своего исходного размера. Второй слой может использовать блоки размером  $64 \times 64$  пикселя без масштабирования. У каждого слоя есть собственный массив карты для работы, так что самая сложная часть — гарантировать, что все они совпадут друг с другом.

## Создание больших блоков

При работе с большими растрами возникает одна проблема — как разделить их на несколько текстур? Предположим, что размер вашего уровня  $1024 \times 1024$  пикселя; это значит, что надо разделить уровень на 16 текстур, каждая размером  $256 \times 256$  пикселей. Поскольку Direct3D не может сделать это, придется воспользоваться помощью другой программы.

**ПРИМЕЧАНИЕ**

Демонстрационная версия Paint Shop Pro (PSP), замечательной программы редактирования изображений, находится на прилагаемом к книге CD-ROM в каталоге \Utils\Paint Shop Pro 8\ . Используя PSP вы сможете разбить растры на небольшие удобные изображения.

Можете разделить на фрагменты свое любимое большое изображение и использовать его в примере из следующего раздела, либо откройте пример, находящийся на CD-ROM (в папке \BookCode\Chap07\Bitmap).

После того, как вы сохраните фрагменты большого растрового изображения в различных файлах, наступит время загрузить их. Для загрузки этих растровых изображений с диска и рисования их на экране вы можете использовать чрезвычайно полезный объект класса **cTiles**, разработанный ранее в этой главе в разделе «Построение класса для работы с блоками». Верно — не требуется никакого программирования; все уже сделано!

Поскольку теперь вы можете загрузить блоки вашего большого растра и можете использовать разработанный ранее класс **cMap** для хранения информации о карте, перейдем к рисованию каких-нибудь больших растров уровней.

## Большой пример

Теперь, когда у вас есть набор больших блоков и набор небольших блоков, пришло время выступить во всем блеске. Фактически, механизм работы с большими растрами, который я описал и разработал для этой книги почти не отличается от механизма работы с блочной графикой, который я показал в этой главе раньше (см. раздел «Основы использования блочной графики»). Поэтому не надо поворачивать колесо вспять и снова показывать тот же самый код. Вместо этого взгляните на демонстрационную программу работы

с большими растрами, находящуюся на CD-ROM. Единственное различие, которое вы заметите, заключается в том, что некоторые слои используют большие блоки.

## Заканчиваем с двухмерной графикой

Теперь вы узнали, как использовать предоставляемые Direct3D возможности двухмерной графики. Это включает использование экономящей память техники блочной графики для создания огромных уровней. Немного потрудившись вы сможете преобразовать представленные в этой главе базовые техники в полнофункциональный игровой движок, готовый для использования в ваших проектах.

Если вы достаточно смелы, почему бы не попробовать принарядить движок для работы с картами, включив в него ряд новых возможностей, таких как анимация блоков? Осуществляется анимация легко; меняя номер блока в массиве карты вы динамически изменяете изображение блока, который рисуется в данном месте. Создав из блоков анимационную последовательность, вы затем сможете последовательно менять номера блоков в массиве карты для показа этой последовательности изображений.

Начиная с главы 8, «Создание трехмерного графического движка», и в нескольких последующих главах я сосредоточусь в основном на использовании трехмерной графики.

### Программы на CD-ROM

Программы, демонстрирующие обсуждавшийся в этой главе код, находятся на прилагаемом к книге CD-ROM. В папке \BookCode\Chap07\ вы найдете следующие программы:

**Tile** — пример работы с блочной графикой, демонстрирующий использование блоков, карт и техники плавной прокрутки.  
Местоположение: \BookCode\Chap07\Tile\.

**Bitmap** — демонстрация работы с большими растрами.  
Местоположение: \BookCode\Chap07\Bitmap\.

# Глава 8

## Создание трехмерного графического движка

В главе 2, «Рисование с DirectX Graphics» вы узнали, как визуализировать трехмерную графику, но работали только с небольшими сетками. Теперь вам нужна возможность визуализировать большие карты и уровни, по которым будут бродить игроки. Вы уже видели, как работать с двухмерными мирами — теперь пришло время перейти к трехмерному миру!

В этой главе вы узнаете следующее:

- Использование сеток в качестве уровней.
- Использование пирамиды видимого пространства.
- Визуализация уровней с использованием узлов и деревьев.
- Добавление в мир трехмерных объектов.
- Обнаружение столкновений.
- Небесный куб.

### Сетки в качестве уровней

Возможно, вы сочтете меня сумасшедшим, но нет ничего проще использования сетки в качестве уровня. В этом случае вы легко можете использовать базовые возможности графического ядра. Недостаток использования единой сетки в том, что вы визуализируете весь уровень сразу, а значит даже невидимые части пройдут через конвейер визуализации, чтобы быть отброшенными. Говоря по-русски, вы будете зря терять время.

Это не должно обескураживать вас, поскольку есть несколько замечательных способов использования одной сетки для визуализации уровня. Предположим, мир вашей игры состоит исключительно из подземелий. Каждое подземелье состоит из различных комнат, соединенных коридорами. Видите, что получается? Каждая комната и коридор — это отдельная сетка; вам надо только загружать и освобождать представляющие комнаты подземелья сетки по ходу игрового процесса.



Чтобы увидеть, о чем я говорю, загрузите демонстрационную программу MeshLvl, находящуюся на прилагаемом к книге CD-ROM (посмотрите в папке BookCode\Chap08\MeshLvl\). Программа показывает загрузку нескольких сеток и их совместное рисование в виде большого уровня. На рис. 8.1 показано окно программы MeshLvl.



*Рис. 8.1. Программа MeshLvl в действии. Обратите внимание, что для рисования различных комнат используются только две сетки*

Вот как работает программа MeshLvl: главная функция (которую мы разберем здесь) загружает две сетки — коридор и комнату. Каждый коридор и комната в игре представлены объектом **cObject** из графического ядра, с которым в свою очередь связана соответствующая сетка. Для визуализации сцены вам необходимо ориентировать коридоры и комнаты относительно местоположения зрителя и нарисовать их, используя функции визуализации сеток, такие как **cObject::Render**. В демонстрационной программе вы можете использовать клавиши управления курсором для перемещения по коридорам и комнатам и мышшь для вращения камеры.

## Загрузка уровней

Как я упоминал в предыдущем разделе, уровни конструируются из различных сеток, которые необходимо загрузить. Для наших целей эти сетки загружаются в набор объектов сеток графического ядра (**cMesh**) и экземпляры класса трехмерных объектов (**cObject**). Для загрузки используемых в программе MeshLvl сеток и их назначения трехмерным объектам применяется следующий код:

```
// Объявления в классе
cMesh m_LevelMeshes[2];
cObject m_LevelObjects[8];

// ... далее в коде инициализации

// Загрузка сеток комнат
m_RoomMeshes[0].Load(&m_Graphics,
                    "..\\LevelData\\Corridor.x",
                    "..\\LevelData\\");
m_RoomMeshes[1].Load(&m_Graphics,
                    "..\\LevelData\\Room.x",
                    "..\\LevelData\\");

// Установка объектов комнат
m_RoomObjects[0].Create(&m_Graphics, &m_RoomMeshes[1]);
m_RoomObjects[1].Create(&m_Graphics, &m_RoomMeshes[0]);
m_RoomObjects[2].Create(&m_Graphics, &m_RoomMeshes[1]);
m_RoomObjects[3].Create(&m_Graphics, &m_RoomMeshes[0]);
m_RoomObjects[4].Create(&m_Graphics, &m_RoomMeshes[0]);
m_RoomObjects[5].Create(&m_Graphics, &m_RoomMeshes[1]);
m_RoomObjects[6].Create(&m_Graphics, &m_RoomMeshes[0]);
m_RoomObjects[7].Create(&m_Graphics, &m_RoomMeshes[1]);
```

В предшествующем коде показана загрузка двух сеток: Corridor.x и Room.x. Для хранения сеток используется пара объектов **cMesh**. Затем создается восемь объектов (каждый из которых является экземпляром **cObject**), представляющих комнаты (или соединяющие их коридоры). Теперь комнаты и коридоры необходимо ориентировать:

```
// m_RoomMeshes[0], [2], [5], [7] используют сетку комнаты (room.x)
// m_RoomMeshes[1], [3], [4], [6] используют сетку коридора (corridor.x)
m_RoomObjects[0].Move(-2000.0f, 0.0f, 2000.0f);
m_RoomObjects[1].Move( 0.0f, 0.0f, 2000.0f);
m_RoomObjects[2].Move( 2000.0f, 0.0f, 2000.0f);
m_RoomObjects[3].Move(-2000.0f, 0.0f, 0.0f);
m_RoomObjects[4].Move( 2000.0f, 0.0f, 0.0f);
m_RoomObjects[5].Move(-2000.0f, 0.0f, -2000.0f);
m_RoomObjects[6].Move( 0.0f, 0.0f, -2000.0f);
m_RoomObjects[7].Move( 2000.0f, 0.0f, -2000.0f);

m_RoomObjects[0].Rotate(0.0f, 0.00f, 0.0f);
m_RoomObjects[1].Rotate(0.0f, 1.57f, 0.0f);
m_RoomObjects[2].Rotate(0.0f, 1.57f, 0.0f);
m_RoomObjects[5].Rotate(0.0f, -1.57f, 0.0f);
m_RoomObjects[6].Rotate(0.0f, -1.57f, 0.0f);
m_RoomObjects[7].Rotate(0.0f, 3.14f, 0.0f);
```

Ну вот! Все готово к рисованию комнат.

## Рисование комнат

После того, как сетки загружены, пришло время визуализировать их на экране. В каждом кадре программа MeshLvl определяет местоположение пользователя в образующих уровень комнатах. После этого она ориентирует и визуализирует каждую представляющую комнату сетку. У нас всего четыре комнаты и четыре соединяющих их коридора. Используется две сетки и значит каждая сетка рисуется четыре раза.

По указанной причине код, визуализирующий представление, получается очень простым. Перебирая в цикле элементы массива, хранящие ориентацию каждой из комнат, программа визуализирует каждую сетку, используя короткую последовательность команд. В приведенном ниже коде вы увидите короткий цикл, занимающийся ориентированием и рисованием комнат. Также обратите внимание, что я добавил код для чтения пользовательского ввода и соответствующего перемещения точки просмотра (с использованием объекта **cCamera**).

Клавиши управления курсором перемещают точку просмотра, а перемещение мыши изменяет направление взгляда. В каждом кадре экранные координаты мыши (по оси X) преобразуются в значение поворота относительно оси Y. При перемещении мыши влево или вправо соответственно влево или вправо поворачивается направление взгляда. Поворот относительно оси Y сохраняется в объекте **cCamera**, представляющем местоположение зрителя в трехмерном мире.

Когда пользователь нажимает клавишу управления курсором, вращение по оси Y используется для перемещения зрителя. Зритель перемещается вперед, назад, влево или вправо на расстояние, определяемое по времени, прошедшему с предыдущего вызова **Frame** до текущего вызова **Frame** (оно хранится в переменной **Elapsed**). Направление перемещения определяется по повороту относительно оси Y, вычисляемому на основе координаты X мыши.

Взгляните на функцию, которая обрабатывает визуализацию и ввод:

```
BOOL cApp::Frame()
{
    static DWORD Timer = timeGetTime();
    unsigned long Elapsed;
    float XMove, ZMove;
    short i;

    // Вычисляем прошедшее время
    // (плюс повышение скорости)
    Elapsed = (timeGetTime() - Timer) * 2;
    Timer = timeGetTime();

    // Получаем ввод
    m_Keyboard.Read();
    m_Mouse.Read();

    // Обрабатываем ввод и обновляем все.
    // ESC завершает программу
    if(m_Keyboard.GetKeyState(KEY_ESC) == TRUE)
        return FALSE;

    // Обработка перемещения
    XMove = ZMove = 0.0f;

    // Обработка ввода с клавиатуры для
    // перемещения зрителя
    if(m_Keyboard.GetKeyState(KEY_UP) == TRUE) {
        XMove = (float)sin(m_Camera.GetYRotation()) * Elapsed;
        ZMove = (float)cos(m_Camera.GetYRotation()) * Elapsed;
    }
}
```

```

if(m_Keyboard.GetKeyState(KEY_DOWN) == TRUE) {
    XMove = -(float)sin(m_Camera.GetYRotation()) * Elapsed;
    ZMove = -(float)cos(m_Camera.GetYRotation()) * Elapsed;
}
if(m_Keyboard.GetKeyState(KEY_LEFT) == TRUE) {
    XMove = (float)sin(m_Camera.GetYRotation() - 1.57f) * Elapsed;
    ZMove = (float)cos(m_Camera.GetYRotation() - 1.57f) * Elapsed;
}
if(m_Keyboard.GetKeyState(KEY_RIGHT) == TRUE) {
    XMove = (float)sin(m_Camera.GetYRotation() + 1.57f) * Elapsed;
    ZMove = (float)cos(m_Camera.GetYRotation() + 1.57f) * Elapsed;
}

// Обновление координат зрителя
m_XPos += XMove;
m_ZPos += ZMove;

// Размещаем камеру и поворачиваем ее
// согласно перемещению мыши
m_Camera.Move(m_XPos + XMove, 400.0f, m_ZPos + ZMove);
m_Camera.RotateRel((float)m_Mouse.GetYDelta() / 200.0f,
                  (float)m_Mouse.GetXDelta() / 200.0f,
                  0.0f);

// Устанавливаем камеру
m_Graphics.SetCamera(&m_Camera);

// Визуализируем все
m_Graphics.Clear(D3DCOLOR_RGBA(0,64,128,255));
if(m_Graphics.BeginScene() == TRUE) {

    // Визуализируем каждую комнату
    for(i = 0; i < 8; i++)
        m_RoomObjects[i].Render();
    m_Graphics.EndScene();
}
m_Graphics.Display();
return TRUE;
}

```

## Усовершенствование базовой техники

Рисовать трехмерные миры не трудно, но появляется одна проблема. Хотя вы можете рисовать столько сеток, сколько хотите (персонажей, объекты и уровни), вы заметите, что ваш трехмерный движок замедляется с появлением каждого нового объекта.

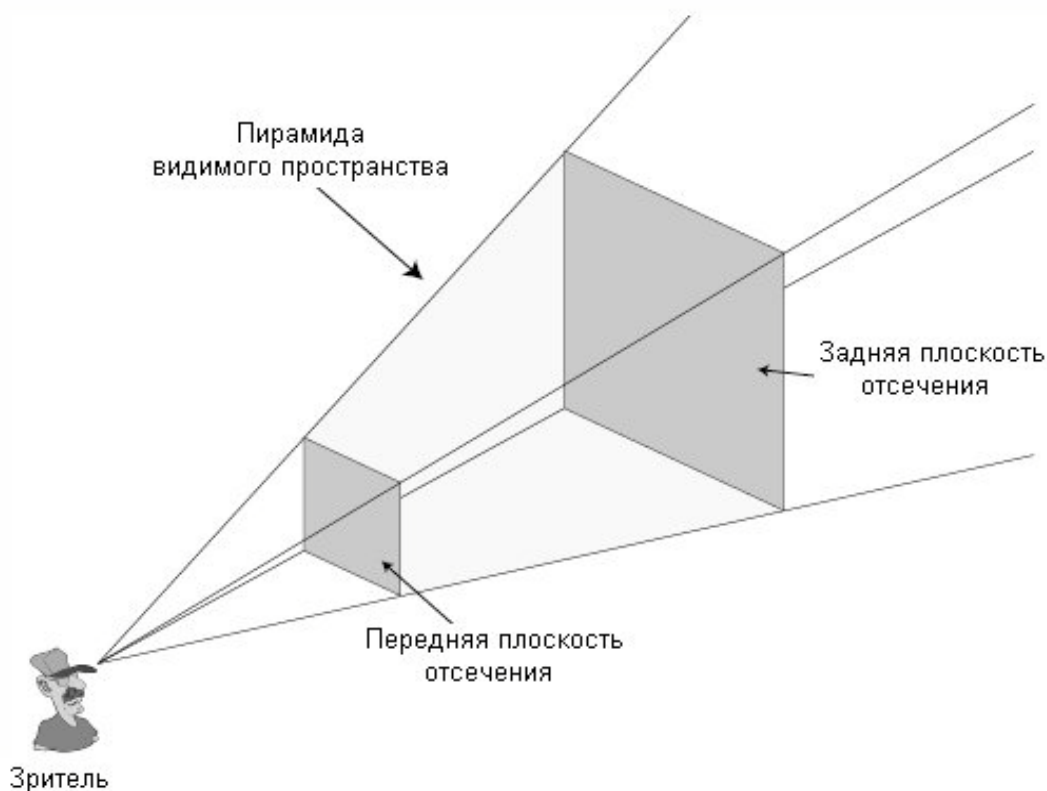
Каждый добавленный в конвейер полигон замедляет всю работу, поскольку его данные необходимо обработать. Это справедливо для всех полигонов любых сеток, включая сетки, которые невидимы (находятся позади вас).

Идеально было бы отбросить сетки, находящиеся вне поля зрения, чтобы Direct3D имел дело только с теми сетками (и полигонами), которые видимы — это немного прибавит скорости. Но как узнать, находится ли что-либо в поле зрения до обработки? Здесь в игру вступает пирамида видимого пространства.

## Знакомство с пирамидой видимого пространства

*Пирамида видимого пространства (viewing frustum)* — это набор из шести плоскостей, расходящихся от точки местоположения зрителя и определяющих, какие полигоны видимы, а какие — нет. Пирамида видимого пространства очень полезна для оптимизации обработки графики, и я хочу познакомить вас со всеми относящимися к ней запутанными деталями.

Вы можете думать о пирамиде видимого пространства, как о расширяющейся в направлении от вас пирамиде, в вершине которой находитесь вы (как показано на рис. 8.2). Эта пирамида представляет ваше *поле зрения (field of view, или FOV)*. Все, что находится внутри пирамиды вы видите, а все, что снаружи, — нет.



**Рис. 8.2.** Пирамида видимого пространства обычно имеет форму усеченной пирамиды. Зритель видит все, что находится внутри нее

Если вы недоумеваете, как пирамида видимого пространства может помочь вам в трехмерном движке, примите во внимание следующее: все в вашем трехмерном графическом движке состоит из точек в трехмерном пространстве (называемых вершины). У пирамиды видимого пространства есть шесть сторон: передняя, задняя, левая, правая, верхняя и нижняя). Используя ряд математических вычислений можно определить, какие вершины находятся внутри пирамиды, а какие — вне ее. Вершины внутри пирамиды визуализируются, а вершины, находящиеся вне ее, — нет. То же самое справедливо и при визуализации полигонов — визуализируются только те вершины или грани, которые расположены внутри пирамиды видимого пространства.

## Плоскости и отсечение

Шесть сторон пирамиды видимого пространства называются *плоскостями отсечения* (*clipping planes*). Для простоты думайте о плоскости как о бесконечно большом листе бумаги (с лицевой и обратной сторонами). Плоскость определяется четырьмя числами, которые обычно называют А, В, С и D. Эти четыре числа определяют ориентацию плоскости в трехмерном пространстве.

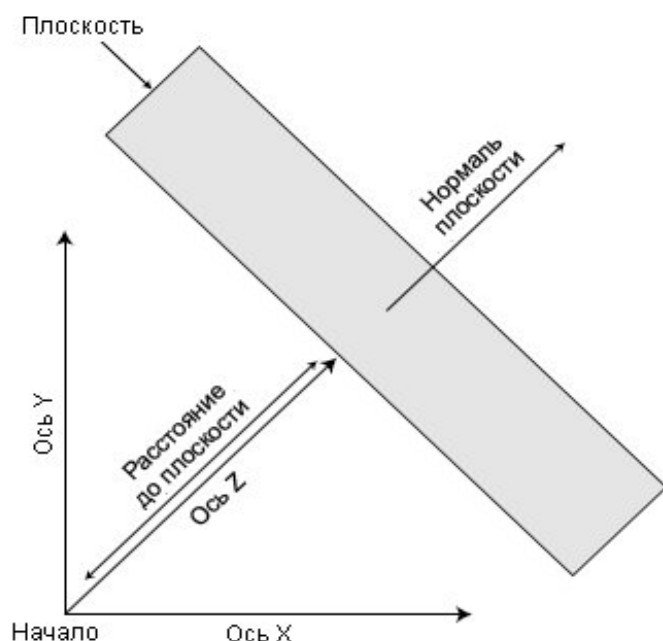
---

**ПРИМЕЧАНИЕ** Direct3D, точнее D3DX, использует для хранения данных плоскости специальный объект `D3DXPLANE`. Он содержит четыре переменные: `a`, `b`, `c` и `d` — все типа `float`.

---

В главе 2 я упоминал нормали — числа, определяющие куда направлен объект. Обычно используемые для освещения, нормали могут применяться для описания ориентации чего угодно, и в нашем случае это направление лицевой стороны плоскости.

Вы определяете плоскость задавая ее ориентацию и перемещая на требуемое расстояние от начала координат. (В действительности плоскость определяется точно так же, как нормаль, но с дополнительным указанием расстояния от начала координат.) Взгляните на рис. 8.3, где показана ориентированная плоскость в трехмерном пространстве.



**Рис. 8.3.** Описание плоскости включает направление ее лицевой стороны и расстояние до начала координат

Вместо того, чтобы задавать значения нормали плоскости как X, Y, Z, вы используете переменные А, В и С. Требуется еще одно дополнительное значение для задания расстояния от плоскости до начала координат. Это расстояние представляется как D. Описывая плоскость вы устанавливаете в переменных А, В и С значения нормали плоскости, а в D — расстояние от плоскости до начала координат. Как только нормаль и расстояние заданы, вы можете использовать плоскость для того, чтобы проверить, находится ли указанная точка перед плоскостью или за ней.

Для вычисления шести плоскостей пирамиды видимого пространства вы комбинируете текущую матрицу преобразования вида и матрицу проекции. Затем вы имеете дело непосредственно с комбинированной матрицей для вычисления значений A, B, C и D каждой плоскости.

Вот как комбинируются две требуемые матрицы и на их основании вычисляются значения плоскостей (значения плоскостей помещаются в соответствующие объекты **D3DXPLANE**):

```
// Graphics = ранее инициализированный объект cGraphics
D3DXPLANE Planes[6]; // Шесть плоскостей
// пирамиды видимого пространства
D3DXMATRIX Matrix, matView, matProj; // Рабочие матрицы

// Получаем матрицы вида и проекции и комбинируем их
Graphics.GetDeviceCOM()->GetTransform(D3DTS_PROJECTION, &matProj);
Graphics.GetDeviceCOM()->GetTransform(D3DTS_VIEW, &matView);
D3DXMatrixMultiply(&Matrix, &matView, &matProj);

// Вычисляем плоскости
Planes[0].a = Matrix._14 + Matrix._13; // Передняя плоскость
Planes[0].b = Matrix._24 + Matrix._23;
Planes[0].c = Matrix._34 + Matrix._33;
Planes[0].d = Matrix._44 + Matrix._43;
D3DXPlaneNormalize(&Planes[0], &Planes[0]);

Planes[1].a = Matrix._14 - Matrix._13; // Задняя плоскость
Planes[1].b = Matrix._24 - Matrix._23;
Planes[1].c = Matrix._34 - Matrix._33;
Planes[1].d = Matrix._44 - Matrix._43;
D3DXPlaneNormalize(&Planes[1], &Planes[1]);

Planes[2].a = Matrix._14 + Matrix._11; // Левая плоскость
Planes[2].b = Matrix._24 + Matrix._21;
Planes[2].c = Matrix._34 + Matrix._31;
Planes[2].d = Matrix._44 + Matrix._41;
D3DXPlaneNormalize(&Planes[2], &Planes[2]);

Planes[3].a = Matrix._14 - Matrix._11; // Правая плоскость
Planes[3].b = Matrix._24 - Matrix._21;
Planes[3].c = Matrix._34 - Matrix._31;
Planes[3].d = Matrix._44 - Matrix._41;
D3DXPlaneNormalize(&Planes[3], &Planes[3]);

Planes[4].a = Matrix._14 - Matrix._12; // Верхняя плоскость
Planes[4].b = Matrix._24 - Matrix._22;
Planes[4].c = Matrix._34 - Matrix._32;
Planes[4].d = Matrix._44 - Matrix._42;
D3DXPlaneNormalize(&Planes[4], &Planes[4]);

Planes[5].a = Matrix._14 + Matrix._12; // Нижняя плоскость
Planes[5].b = Matrix._24 + Matrix._22;
Planes[5].c = Matrix._34 + Matrix._32;
Planes[5].d = Matrix._44 + Matrix._42;
D3DXPlaneNormalize(&Planes[5], &Planes[5]);
```

**ПРИМЕЧАНИЕ**

Обратите внимание, что каждая плоскость нормализуется (в данном случае с использованием функции **D3DXPlaneNormalize**), чтобы гарантировать, что значения **a**, **b**, **c** будут меньше или равны 1.0f, а значение **d** будет хранить расстояние от плоскости до начала координат.

## **Проверка видимости с плоскостями**

Теперь у вас есть плоскость (или набор плоскостей), ориентированная в заданном направлении. Чтобы проверить, находится ли точка перед плоскостью или за ней, вы вычисляете *скалярное произведение* (*dot product*). Скалярное произведение — это специальная операция с векторами (координатами), обычно применяемая для вычисления угла между двумя векторами.

При проверке местоположения точки относительно плоскости скалярное произведение сообщает вам расстояние от точки до плоскости. Если значение положительно, точка находится перед плоскостью. Если значение отрицательно — точка находится за плоскостью.

Для вычисления скалярного произведения вы используете функцию **D3DXPlaneDotCoord**:

```
FLOAT D3DXPlaneDotCoord(
    CONST D3DXPLANE *pP, // D3DXPLANE для проверки
    CONST D3DXVECTOR3 *pV); // Проверяемая точка
```

Функции **D3DXPlaneDotCoord** вы предоставляете структуру плоскости (содержащую значения плоскости) и точку (вектор, содержащийся в объекте **D3DXVECTOR3**). Проверка определит с какой стороны плоскости находится точка — спереди или сзади. После возвращения из функции **D3DXPlaneDotCoord** вы получаете расстояние от точки до плоскости. Это значение может быть равно нулю, положительным или отрицательным.

Если возвращенное значение равно 0, точка лежит на плоскости. Если значение отрицательно, то точка находится за плоскостью; если положительно — перед ней. Вот пример проверки:

```
// Plane = ранее инициализированный объект D3DXPLANE
// XPos, YPos, ZPos = проверяемая точка
float Dist = D3DXPlaneDotCoord(&Plane,
                                &D3DXVECTOR3(XPos, YPos, ZPos));

// Если dist > 0 точка перед плоскостью
// Если dist < 0 точка за плоскостью
// Если dist == 0 точка на плоскости
```

## **Проверка всей пирамиды**

Хотя проверка отдельной точки полезна, вы можете пойти дальше и проверять целые объекты на нахождение внутри пирамиды видимого пространства. Проверяемыми объектами могут быть кубы, параллелепипеды и сферы.



Для кубов и параллелепипедов вы проверяете угловые вершины. Если все вершины находятся за плоскостью, значит куб или параллелепипед вне пирамиды видимого пространства (и вне поля зрения). Если хоть одна вершина внутри пирамиды, или перед любой плоскостью (не все вершины расположены позади одной плоскости), означает, что куб или параллелепипед видимы. Что касается сферы, то чтобы она была видима расстояние от каждой плоскости до центра сферы должно быть равно или больше радиуса.

Чтобы проверить произвольное количество точек, вы проверяете каждую из них индивидуально, убеждаясь, что хотя бы одна находится перед всеми плоскостями. В следующем разделе вы узнаете как создать класс, конструирующий пирамиду видимого пространства и проверяющий видимость объектов с использованием этой пирамиды.

## Класс cFrustum

Теперь вы познакомились с математикой, лежащей в основе использования пирамиды видимого пространства и, хотя эта математика относительно проста, желательно использовать пирамиду видимого пространства без постоянного повторного набора одного и того же кода. Поскольку математика при каждом использовании пирамиды видимого пространства остается одной и той же, имеет смысл создать класс, который будет выполнять для вас все вычисления, включая вычисление пирамиды видимого пространства и использование пирамиды для проверки видимости объектов.

Класс, о котором я говорю, — это **cFrustum**.

```
class cFrustum
{
private:
    D3DXPLANE m_Planes[6]; // Плоскости пирамиды

public:
    // Конструируем шесть плоскостей на основании
    // текущего вида и проекции. Можете переопределить
    // подставляемое по умолчанию значение глубины.
    BOOL Construct(cGraphics *Graphics, float ZDistance = 0.0f);

    // Последующие функции проверяют отдельную точку, куб,
    // параллелепипед и сферу на предмет нахождения внутри пирамиды.
    // Возвращаемое значение TRUE означает, что объект видим,
    // FALSE - что невидим. При проверке кубов и параллелепипедов
    // вы можете указать переменную BOOL, определяющую все ли
    // точки объекта находятся внутри пирамиды
    BOOL CheckPoint(float XPos, float YPos, float ZPos);
    BOOL CheckCube(float XCenter, float YCenter,
                   float ZCenter, float Size,
                   BOOL *CompletelyContained = NULL);
    BOOL CheckRectangle(float XCenter, float YCenter,
                       float ZCenter, float XSize,
                       float YSize, float ZSize,
                       BOOL *CompletelyContained = NULL);
    BOOL CheckSphere(float XCenter, float YCenter,
                    float ZCenter, float Radius);
};
```

Построенный только из пяти функций и массива объектов **D3DPLANE**, класс **cFrustum** предлагает вам много функциональных возможностей. Давайте взглянем, что можно делать с каждой из этих пяти функций.

### ***cFrustum::Construct***

Вы используете эту функцию, которую необходимо вызывать каждый раз после изменения матриц проекции или вида, для конструирования шести плоскостей проверки:

```

BOOL cFrustum::Construct(cGraphics *Graphics, float ZDistance)
{
    D3DXMATRIX Matrix, matView, matProj;
    float ZMin, Q;

    // Проверка ошибок
    if(Graphics == NULL || Graphics->GetDeviceCOM() == NULL)
        return FALSE;

    // Вычисление данных FOV
    Graphics->GetDeviceCOM()->GetTransform(D3DTS_PROJECTION,
                                           &matProj);

    if(ZDistance != 0.0f) {

        // Вычисление новой матрицы проекции
        // на основании заданной дистанции
        ZMin = -matProj._43 / matProj._33;
        Q = ZDistance / (ZDistance - ZMin);
        matProj._33 = Q;
        matProj._43 = -Q * ZMin;
    }
    Graphics->GetDeviceCOM()->GetTransform(D3DTS_VIEW, &matView);
    D3DXMatrixMultiply(&Matrix, &matView, &matProj);

    // Вычисление плоскостей
    m_Planes[0].a = Matrix._14 + Matrix._13; // Передняя плоскость
    m_Planes[0].b = Matrix._24 + Matrix._23;
    m_Planes[0].c = Matrix._34 + Matrix._33;
    m_Planes[0].d = Matrix._44 + Matrix._43;
    D3DXPlaneNormalize(&m_Planes[0], &m_Planes[0]);

    m_Planes[1].a = Matrix._14 - Matrix._13; // Задняя плоскость
    m_Planes[1].b = Matrix._24 - Matrix._23;
    m_Planes[1].c = Matrix._34 - Matrix._33;
    m_Planes[1].d = Matrix._44 - Matrix._43;
    D3DXPlaneNormalize(&m_Planes[1], &m_Planes[1]);

    m_Planes[2].a = Matrix._14 + Matrix._11; // Левая плоскость
    m_Planes[2].b = Matrix._24 + Matrix._21;
    m_Planes[2].c = Matrix._34 + Matrix._31;
    m_Planes[2].d = Matrix._44 + Matrix._41;
    D3DXPlaneNormalize(&m_Planes[2], &m_Planes[2]);

    m_Planes[3].a = Matrix._14 - Matrix._11; // Правая плоскость
    m_Planes[3].b = Matrix._24 - Matrix._21;
    m_Planes[3].c = Matrix._34 - Matrix._31;
    m_Planes[3].d = Matrix._44 - Matrix._41;
    D3DXPlaneNormalize(&m_Planes[3], &m_Planes[3]);
}

```

```
m_Planes[4].a = Matrix._14 - Matrix._12; // Верхняя плоскость
m_Planes[4].b = Matrix._24 - Matrix._22;
m_Planes[4].c = Matrix._34 - Matrix._32;
m_Planes[4].d = Matrix._44 - Matrix._42;
D3DXPlaneNormalize(&m_Planes[4], &m_Planes[4]);

m_Planes[5].a = Matrix._14 + Matrix._12; // Нижняя плоскость
m_Planes[5].b = Matrix._24 + Matrix._22;
m_Planes[5].c = Matrix._34 + Matrix._32;
m_Planes[5].d = Matrix._44 + Matrix._42;
D3DXPlaneNormalize(&m_Planes[5], &m_Planes[5]);

return TRUE;
}
```

Большую часть этого кода вы уже видели в разделе «Знакомство с пирамидой видимого пространства» этой главы, но здесь добавлена одна вещь — возможность переопределять расстояние до дальней плоскости отсечения при формировании пирамиды видимого пространства. Если вы хотите, чтобы видимы были только близкие объекты, задайте новое расстояние до дальней плоскости отсечения. Чтобы вычислить новую плоскость отсечения необходимо заново вычислить значения матрицы для переднего и заднего планов, как показано в коде.

### ***cFrustum::CheckPoint, CheckCube, CheckRectangle и CheckSphere***

Вы используете эти четыре функции, чтобы определить видимо ли что-нибудь внутри пирамиды видимого пространства. Как видите, функции прямолинейны и самодокументируемы:

```
BOOL cFrustum::CheckPoint(float XPos, float YPos, float ZPos)
{
    // Убеждаемся, что точка внутри пирамиды
    for(short i = 0; i < 6; i++) {
        if(D3DXPlaneDotCoord(&m_Planes[i],
            &D3DXVECTOR3(XPos, YPos, ZPos)) < 0.0f)
            return FALSE;
    }
    return TRUE;
}
```

В показанной функции **CheckPoint** проверяется, что рассматриваемая точка находится перед всеми шестью плоскостями отсечения. Помните, что точка находится перед плоскостью, если скалярное произведение положительно. Если для какой-нибудь плоскости скалярное произведение отрицательно, функция возвращает **FALSE**, сообщая, что точка находится за плоскостью и, следовательно, вне поля зрения.

```
BOOL cFrustum::CheckCube(float XCenter, float YCenter,
    float ZCenter, float Size,
    BOOL *CompletelyContained)
{
    short i;
    DWORD TotalIn = 0;
```

```

// Подсчитываем количество точек внутри пирамиды
for(i = 0; i < 6; i++) {
    DWORD Count = 8;
    BOOL PointIn = TRUE;

    // Проверяем все восемь точек относительно плоскости
    if(D3DXPlaneDotCoord(&m_Planes[i],
                        &D3DXVECTOR3(XCenter-Size,
                                      YCenter-Size,
                                      ZCenter-Size)) < 0.0f) {

        PointIn = FALSE;
        Count--;
    }
    if(D3DXPlaneDotCoord(&m_Planes[i],
                        &D3DXVECTOR3(XCenter+Size,
                                      YCenter-Size,
                                      ZCenter-Size)) < 0.0f) {

        PointIn = FALSE;
        Count--;
    }
    if(D3DXPlaneDotCoord(&m_Planes[i],
                        &D3DXVECTOR3(XCenter-Size,
                                      YCenter+Size,
                                      ZCenter-Size)) < 0.0f) {

        PointIn = FALSE;
        Count--;
    }
    if(D3DXPlaneDotCoord(&m_Planes[i],
                        &D3DXVECTOR3(XCenter+Size,
                                      YCenter+Size,
                                      ZCenter-Size)) < 0.0f) {

        PointIn = FALSE;
        Count--;
    }
    if(D3DXPlaneDotCoord(&m_Planes[i],
                        &D3DXVECTOR3(XCenter-Size,
                                      YCenter-Size,
                                      ZCenter+Size)) < 0.0f) {

        PointIn = FALSE;
        Count--;
    }
    if(D3DXPlaneDotCoord(&m_Planes[i],
                        &D3DXVECTOR3(XCenter+Size,
                                      YCenter-Size,
                                      ZCenter+Size)) < 0.0f) {

        PointIn = FALSE;
        Count--;
    }
    if(D3DXPlaneDotCoord(&m_Planes[i],
                        &D3DXVECTOR3(XCenter-Size,
                                      YCenter+Size,
                                      ZCenter+Size)) < 0.0f) {

        PointIn = FALSE;
        Count--;
    }
    if(D3DXPlaneDotCoord(&m_Planes[i],
                        &D3DXVECTOR3(XCenter+Size,
                                      YCenter+Size,
                                      ZCenter+Size)) < 0.0f) {

        PointIn = FALSE;
        Count--;
    }
}

```

```
// Если внутри пирамиды ничего нет, возвращаем FALSE
if (Count == 0)
    return FALSE;

// Обновляем счетчик, если они все перед плоскостью
TotalIn += (PointIn == TRUE) ? 1:0;
}
// Сохраняем флаг BOOL если проверка полного нахождения внутри
if (CompletelyContained != NULL)
    *CompletelyContained = (TotalIn == 6) ? TRUE:FALSE;

return TRUE;
}
```

Тем же способом, которым функция **CheckPoint** проверяет местоположение единственной точки относительно всех шести плоскостей пирамиды видимого пространства, функция **CheckCube** проверяет восемь вершин куба. Поскольку куб симметричен, вы задаете координаты центра и расстояние от центра до одного из краев (можете считать, что задаете радиус куба).

Следует отметить одну вещь относительно **CheckCube** (и следующей функции **CheckRectangle**) — она проверяет местоположение всех восьми точек относительно всех шести плоскостей. Если все точки находятся за одной из плоскостей, функция возвращает **FALSE**, сообщая, что куб полностью вне поля зрения. Если хотя бы одна точка находится внутри пирамиды видимого пространства или пирамида находится внутри куба (внутри пирамиды видимого пространства не содержится вершин, но как минимум одна вершина находится перед плоскостью), функция возвращает **TRUE**, сигнализируя, что куб видим в пирамиде.

Если все точки находятся перед всеми шестью плоскостями, куб полностью находится внутри пирамиды. Здесь в дело вступает переменная **CompletelyContained** типа **BOOL**; она устанавливается в **TRUE**, если все восемь точек находятся внутри пирамиды видимого пространства, в ином случае ей присваивается значение **FALSE**.

Функция **CheckRectangle** идентична функции **CheckCube**, за исключением того, что вы задаете ширину, высоту и глубину отдельно. Помните, что передаваемые функции значения должны быть в два раза меньше, чем реальные ширина, высота и глубина. Например, для параллелепипеда размерами  $30 \times 20 \times 10$  в вызове функции **CheckRectangle** вы указываете 15, 10 и 5.

Поскольку код во многом тот же самый, я пропущу его, и вместо этого приведу только прототип (полный код находится на прилагаемом к книге CD-ROM):

```
BOOL cFrustum::CheckRectangle(float XCenter, float YCenter,
                              float ZCenter, float XSize,
                              float YSize, float ZSize,
                              BOOL *CompletelyContained);
```

Переходим к следующему, и последнему объекту, видимость которого можно определять — сфере. Функция **CheckSphere** слегка отличается от остальных — она получает только координаты центра и радиус сферы:

```

BOOL cFrustum::CheckSphere(float XCenter, float YCenter,
                           float ZCenter, float Radius)
{
    short i;

    // Убеждаемся, что радиус внутри пирамиды
    for(i = 0; i < 6; i++) {
        if(D3DXPlaneDotCoord(&m_Planes[i],
                            &D3DXVECTOR3(XCenter, YCenter, ZCenter)) < -Radius)
            return FALSE;
    }
    return TRUE;
}

```

Показанная функция **CheckSphere**, пожалуй, самая простая функция проверки пирамиды видимого пространства. Если сфера находится перед плоскостью (или касается ее), расстояние до центра будет больше, чем отрицательный радиус. Если расстояние меньше отрицательного радиуса, сфера находится вне поля зрения.

Как видите, у класса **cFrustum** много применений, включая разработку профессионального трехмерного движка, который можно использовать для рисования уровней, что и будет нашей следующей темой.

## Разработка профессионального трехмерного движка

Визуализация уровня как единой сетки очень проста, хотя и несколько неэффективна по времени. Но что делать в тех случаях, когда уровень требует большей детализации — зданий, деревьев, пещер и других необходимых для игры объектов? А как насчет того, чтобы немного упростить разработку уровней? Разве не было бы здорово рисовать уровни в программе трехмерного моделирования, такой как 3D Studio Max (также известной как 3DS Max) или MilkShape 3D, а затем помещать их в вашу игру?

Проблемы с большими детализированными уровнями возникают из-за количества полигонов с которыми приходится иметь дело. Рисование всех полигонов в каждом кадре неэффективно. Для увеличения скорости вы можете визуализировать только те полигоны, которые находятся в поле зрения, а чтобы игра работала еще быстрее, исключить сканирование каждого полигона сцены, определяющее видим ли он.

Как можно определить, какие полигоны видимы, не сканируя их все в каждом кадре? Решение заключается в разделении трехмерной модели (представляющей уровень) на небольшие фрагменты (называемые *узлами*, *nodes*), содержащие несколько полигонов. Затем вы упорядочиваете узлы в специальную структуру (*дерево*, *tree*) которую можно быстро

просканировать для определения того, какие узлы видимы. Потом вы визуализируете видимые узлы.

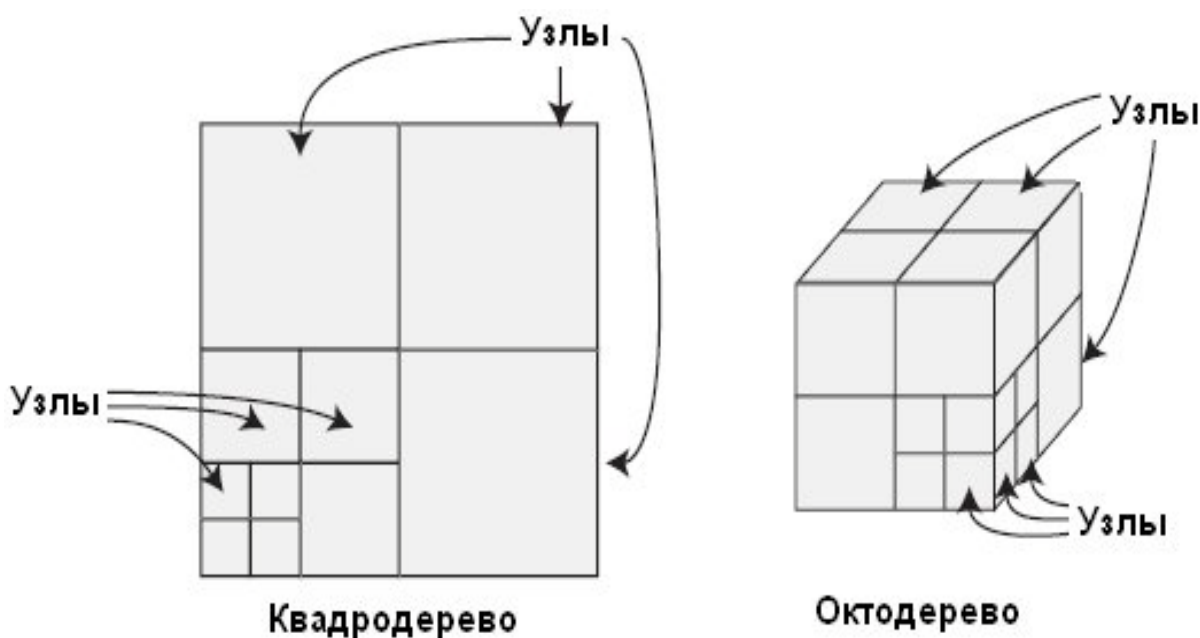
Вы можете определить, какие узлы видимы, используя пирамиду видимого пространства. Теперь вместо того, чтобы сканировать тысячи полигонов, вы сканируете небольшой набор узлов, чтобы определить, что рисовать. Видите, насколько просто это улучшение процесса рисования?

Видю, вы начинаете беспокоиться о том, как осуществить столь выдающийся подвиг. Позвольте представить вам движок NodeTree. Созданный специально для этой книги, он может взять любую сетку и разделить ее на узлы, используемые для быстрой визуализации сеток (например сеток, используемых в качестве уровней вашей игры).

## Знакомство с движком NodeTree

Движок NodeTree достаточно универсален, поскольку может работать в двух различных режимах (отличающихся способом разделения на узлы): режиме *квадродерева* (*quadtree*) и режиме *октодерева* (*octree*). В режиме квадродерева мир (и последующие узлы) делится на четыре узла. Этот режим лучше подходит для сеток уровней, где нет значительных изменений по оси Y (высота точки просмотра меняется не сильно).

Режим октодерева разделяет мир (и последующие узлы) на восемь узлов. Используйте этот режим для больших трехмерных сеток, в которых точка просмотра может располагаться в любом месте. Процесс деления продемонстрирован на рис. 8.4.



**Рис. 8.4.** Режим квадродерева разделяет мир (и последующие узлы) на четыре узла за раз, а режим октодерева разделяет мир (и последующие узлы) на восемь узлов за раз

**ПРИМЕЧАНИЕ**

Выбор используемого режима разделения зависит от вас. Примите во внимание вашу сетку — у вас есть замок для прочесывания, пещеры для раскопок или ландшафт для путешествий? Если высоты вершин сеток не слишком различаются (как, например, в ландшафте), лучше подойдет режим квадродерева. С другой стороны, если сетка распространяется по всем осям (например, замок с несколькими уровнями), используйте режим октодерева.

Мир (представляемый как куб, заключающий в себе все полигоны, описывающие уровень) разделяется на меньшие узлы равного размера. Квадродерево разделяет узлы в двухмерном пространстве (используя оси X и Z), а октодерево разделяет узлы в трехмерном пространстве (используя все оси).

Я вернусь к разделению узлов чуть позже, а перед тем, как продолжить, хочу прояснить несколько моментов. Узел представляет группу полигонов и в то же время представляет область трехмерного пространства. Каждый узел связан с другими отношениями родитель-потомок. Это означает, что узел может содержать другие узлы, и каждый последующий узел является частью, меньшей чем его родитель. Весь трехмерный мир в целом считают *корневым узлом* (*root node*) — самым верхним узлом, с которым связаны все другие узлы.

**ПРИМЕЧАНИЕ**

Чтобы лучше разобраться с узлами, представьте себе Кубик Рубика. Весь куб в целом — это корневой узел, а каждый цветной фрагмент куба — дочерний узел. Каждому дочернему узлу назначены несколько цветов, или полигонов. Эти полигоны (цвета) относятся к узлу (фрагменту куба), который в свою очередь относится к узлу более высокого уровня (целому кубу).

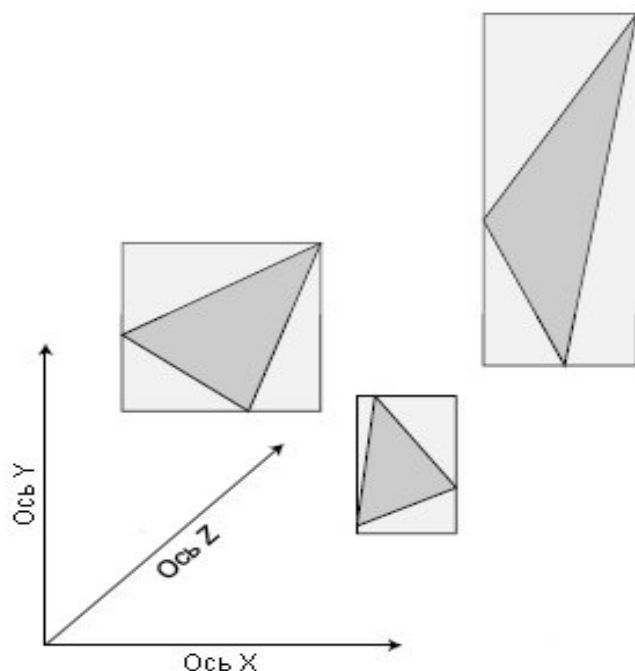
Вот трюк для узлов и деревьев: зная, какие полигоны содержатся в узле трехмерного пространства, вы можете сгруппировать их; затем, начав с корневого узла, вы сможете быстро перебрать каждый узел дерева.

## **Создание узлов и деревьев**

Чтобы создать узлы и построить древовидную структуру вы проверяете каждый полигон сетки. Не беспокойтесь; это делается только один раз и влияние этого процесса на быстродействие можно не учитывать. Ваша цель — решить, как упорядочивать узлы в дереве.

Каждый полигон сетки заключается в прямоугольник (называемый ограничивающим прямоугольником, как показано на рис. 8.5). Прямоугольник определяет протяженность полигона по всем направлениям. Если ограничивающий прямоугольник полигона находится в узле трехмерного пространства (полностью или частично), значит полигон относится к узлу. Полигон может относиться к нескольким узлам, поскольку протяженность полигона может распространяться на несколько узлов.





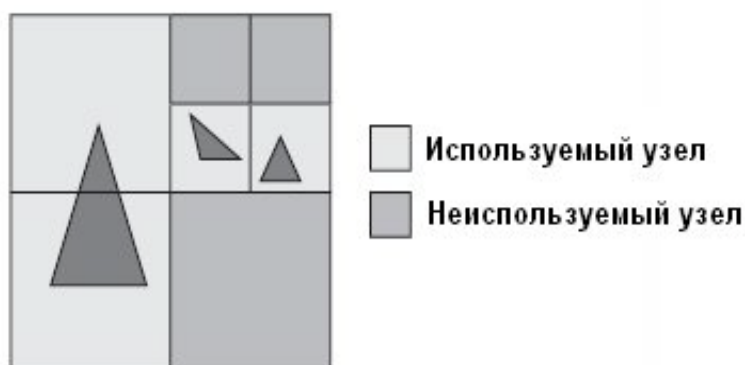
**Рис. 8.5.** У полигонов есть воображаемые (ограничивающие) прямоугольники, окружающие их. Ограничивающие прямоугольники полезны для быстрого определения местоположения полигона в трехмерном пространстве

В процессе группировки полигонов в узлы обращайте внимание, есть ли большие пространства между полигонами или много полигонов сосредоточено в одном месте. В последнем случае необходимо разделить узел на более мелкие подузлы и заново просканировать список полигонов, приняв во внимание новые узлы. Продолжайте этот процесс, пока все полигоны не будут разделены на достаточно маленькие группы и пока каждый узел, содержащий полигоны, не станет достаточно маленьким.

#### СОВЕТ

Для оптимизации древовидной структуры отбрасывайте все узлы не содержащие полигонов. Отбрасывание пустых узлов экономит память и позволяет быстро пересекать дерево. Знайте, что экономия памяти и времени это залог успеха.

На рис. 8.6 показано несколько полигонов и мир (корневой узел), заключенный в квадрат. Квадрат разделен на меньшие узлы, а те, в свою очередь, также разделены на узлы. Каждый узел либо используется (содержит полигон или полигоны), либо не используется (не содержит полигонов).



**Рис. 8.6.** Пример мира с несколькими полигонами, сгруппированными в узлы

Поскольку полигоны на рис. 8.6 расположены далеко друг от друга, вы можете разделить корневой узел на четыре меньших узла (создав квадродерево). Затем проверьте каждый узел и продолжайте разделять большие. Для ускорения процесса можете пропустить пустые узлы. В конце концов вы получите замечательную древовидную структуру, готовую для сканирования.

### **Сканирование и рисование дерева**

Построив древовидную структуру вы готовы визуализировать ее. Начав с корневого узла, вы выполняете проверку попадания узла в пирамиду видимого пространства. Узел считается видимым, если хотя бы одна из восьми точек (думайте о них, как об углах куба), формирующих его в трехмерном пространстве, находится внутри пирамиды видимого пространства или сама пирамида видимого пространства находится внутри узла.

После того, как вы решили, что узел видим, выполните ту же самую проверку для его дочерних узлов (если они есть). Если у узла нет потомков, проверьте, есть ли в нем полигоны, которые надо рисовать (дочерние узлы в процессе сканирования могут рисовать одни и те же полигоны). После того, как полигоны в узле нарисованы, они отмечаются, как нарисованные, и сканирование возвращается к родительскому узлу (и любым другим оставшимся узлам, которые будут просканированы).

---

**ПРИМЕЧАНИЕ**

Если узел полностью находится внутри пирамиды видимого пространства, нет необходимости сканировать его потомков, поскольку они тоже будут полностью находиться внутри пирамиды видимого пространства.

---

Видите магию этого процесса: высокоуровневые узлы отбрасываются вместе с их потомками. Этот способ позволяет удалить из процесса визуализации тысячи полигонов и сэкономить время.

Когда вы будете работать с древовидными структурами и Direct3D, то обнаружите, что сетки содержат много материалов. Помните, что переключение материала это ресурсоемкая и медленная операция (особенно, когда каждый материал использует свою текстуру), так что здесь следует вести себя осмотрительно. Как можно нарисовать все видимые полигоны не переключая материалы снова и снова (даже возвращаясь к уже использованным материалам)? Здесь в игру вступают группы материалов.

## **Работа с группами материалов**

*Группа материалов (material group)* — это набор полигонов, объединенных вместе на основании назначенных материалов. Поскольку сетка может содержать много материалов, имеет смысл визуализировать только те группы полигонов, которым назначен материал, установленный в данное время. В этом случае вы устанавливаете каждый материал (и соответствующую текстуру, если она существует) только один раз,

визуализируете полигоны, использующие этот материал, и переходите к следующему материалу.

Хотя использование групп материалов выглядит логичным, группировка полигонов по их материалам усложняет работу со структурой NodeTree. Вы не можете знать, какие полигоны будут нарисованы не просканировав древовидную структуру. Необходимо просканировать дерево и построить список визуализируемых полигонов. Завершив сканирование просто проверьте список полигонов, относящихся к каждому из материалов, и работайте с ним.

Группировка материалов оказывает влияние только на порядок, в котором рисуются видимые полигоны. Симпатично, да?

## Создание класса cNodeTreeMesh

Пришло время для программирования. Начнем с создания нескольких структур, представляющих вершины, полигоны, узлы и группы материалов:

```
// sVertex - это структура настраиваемого формата вершин,
// содержащая только трехмерные координаты. Она используется
// для получения информации о координатах из буфера вершин сетки
typedef struct sVertex {
    float x, y, z; // Трехмерные координаты
} sVertex;

// Структура данных полигона хранит группу материалов (ее номер),
// номер кадра, в котором полигон рисовался последний раз
// (чтобы в одном кадре не перерисовывать его несколько раз)
// и три вершины, используемые для визуализации полигона
// (о которой вы прочтете позже)
typedef struct sPolygon {
    unsigned long Group; // Группа материалов
    unsigned long Timer; // Кадр последнего рисования
    unsigned short Vertex0; // Индекс вершины 1
    unsigned short Vertex1; // Индекс вершины 2
    unsigned short Vertex2; // Индекс вершины 3
    sPolygon() { Group = Timer = 0; }
} sPolygon;

// Структура данных узла хранит счетчик количества полигонов в узле,
// массив структур sPolygon, трехмерные координаты узла
// (а также размер от центра до ребра, поскольку наши узлы будут
// идеальными кубами) и указатель на дочерние узлы
typedef struct sNode {
    float XPos, YPos, ZPos; // Координаты центра узла
    float Size; // Размер узла
    unsigned long NumPolygons; // Количество полигонов в узле
    unsigned long *PolygonList; // Список полигонов
    sNode *Nodes[8]; // Дочерние узлы 4 для квадродерева,
                    // 8 для октодерева
    // Конструктор, используемый для очистки переменных
    sNode()
    {
        XPos = YPos = ZPos = Size = 0.0f; // Местоположение и размер
        NumPolygons = 0; // Полигонов в узле нет
        PolygonList = NULL; // Очистка списка полигонов
    }
}
```

```

        for(short i = 0; i < 8; i++)          // Квадродерево использует
                                                // только первые 4
            Nodes[i] = NULL;                  // Очищаем указатели
                                                // на дочерние узлы
    }

    // Деструктор для очистки дочерних узлов и переменных
    ~sNode()
    {
        delete [] PolygonList; // Удаляем список полигонов
        PolygonList = NULL;
        for(short i = 0; i < 8; i++) {
            delete Nodes[i];    // Удаляем дочерние узлы
            Nodes[i] = NULL;
        }
    }
} sNode;

// Структура данных группы материалов используется как
// буфер индексов для хранения полигонов, которые необходимо
// визуализировать в одном кадре, а также содержит количество полигонов
// в группе материалов и как много полигонов рисуется в каждом кадре
typedef struct sGroup {
    unsigned long NumPolygons;          // Количество полигонов в группе
    unsigned long NumPolygonsToDraw;    // Количество рисуемых полигонов
    IDirect3DIndexBuffer9 *IndexBuffer;
    unsigned short *IndexPtr;

    // Очистка количества полигонов
    sGroup() { NumPolygons = 0; IndexBuffer = NULL; }

    // Освобождение буфера индексов
    ~sGroup()
    {
        if(IndexBuffer != NULL)
            IndexBuffer->Release();
        IndexBuffer = NULL;
    }
} sGroup;

```

Каждая структура хранит различные виды информации о сетке или структуре `NodeTree`. Базовой структурой данных настраиваемого формата вершин является **sVertex**; она напрямую отображается на все другие структуры данных вершин. Если вы загружаете сетку с диска, то должны использовать структуру **sVertex** для получения координат вершин.

Структура **sPolygon** используется для хранения информации о каждом полигоне сцены. Структура хранит номер группы материалов (материалы нумеруются от нуля до количества материалов минус один), номер кадра в котором последний раз выполнялось рисование (для предотвращения перерисовок) и три вершины, используемые для рисования полигона.

Структура **sNode** хранит количество полигонов, находящихся в узле трехмерного пространства. У каждого узла есть координаты, определяющие местоположение его центра, а также переменная размера, сообщающая расстояние от центра узла до одной из его граней (все узлы — идеальные

кубы). Массив **PolygonList** вы используете для хранения индексов массива структур **sPolygon** (который содержит информацию о каждом полигоне в NodeTree). Также присутствуют восемь указателей на дочерние узлы, применяемые для формирования древовидной структуры. Обратите внимание, что квадродерево использует только четыре первых указателя на дочерние узлы.

Последняя структура, **sGroup**, управляет группами материалов. Вот здесь можно запутаться, главным образом потому, что я забыл познакомить вас в главе 2 с объектом **IDirect3DIndexBuffer9**. При создании NodeTree вам потребуется столько структур **sGroup**, сколько материалов находится в исходной сетке вашего дерева.

Каждая структура **sGroup** содержит информацию об отдельной группе материалов, такую как количество полигонов, использующих данный материал и количество рисуемых в кадре полигонов, которые используют данный материал. После построения списка рисуемых полигонов вам понадобится построить список буферов вершин, используемых для визуализации полигонов. Здесь-то и появляется объект **IDirect3DIndexBuffer9**.

В Direct3D есть два способа рисования полигонов — с использованием только буфера вершин и с использованием комбинации буфера вершин и буфера индексов. Вы уже читали о буфере вершин в главе 2; он работает путем назначения трехмерных координат вершин и их преобразования в полигоны на экране. Но что, если вы хотите использовать одну и ту же вершину снова и снова для рисования нескольких полигонов не тратя время и память на многократное сохранение данных одной вершины?

Здесь вам поможет буфер индексов. Помните, как в главе 2 создавая вершины мы нумеровали их? Эти номера называются *индексы вершин* (*vertex indices*). Так что вместо того, чтобы рисовать полигон задавая набор координат вершин, можно указать индексы вершин.

Например, установив буфер вершин, содержащий четыре вершины, можно создать буфер индексов для рисования двух полигонов, указав в нем, что первый полигон использует вершины 0, 1 и 2, а второй использует вершины 1, 2 и 3. Как видите, у нас по-прежнему есть буфер вершин, хранящий их координаты. Но, благодаря использованию буфера индексов, я могу использовать вершины 1 и 2 несколько раз. Это означает, что мне достаточно определить только четыре вершины, а не шесть (как было бы если бы использовался только буфер вершин).

Сам буфер индексов используется для хранения индексов вершин для рисования каждого кадра; поскольку все узлы сканируются в ходе визуализации, все видимые полигоны добавляются в список полигонов в различных группах материалов. Как только все узлы просканированы, каждая группа материалов рисуется с использованием ее буфера индексов.

Хватит говорить о структурах данных; пришло время перейти к коду класса. Взгляните на приведенный ниже класс, который берет сетку и

преобразует ее в древовидную структуру по вашему выбору — либо в квадродерево, либо в октодерево:

```
// Перечисление типов древовидной структуры
enum TreeTypes { QUADTREE = 0, OCTREE };

class cNodeTreeMesh
{
private:

    // Здесь помещаются показанные ранее объявления
    // структур sVertex, sPolygon, sNode и sGroup

    int m_TreeType;                // Тип дерева
                                    // QUADTREE или OCTREE
    cGraphics *m_Graphics;         // Родительский объект cGraphics
    cFrustum *m_Frustum;          // Пирамида видимого пространства

    float m_Size;                  // Размер мирового куба
    float m_MaxSize;              // Максимальный размер узла

    sNode *m_ParentNode;           // Родитель связанного списка узлов

    unsigned long m_NumGroups;     // Количество групп материалов
    sGroup *m_Groups;             // Группы материалов

    unsigned long m_MaxPolygons;  // Максимальное количество
                                    // полигонов в узле

    unsigned long m_NumPolygons;  // Количество полигонов в сцене
    sPolygon *m_Polygons;         // Список полигонов

    unsigned long m_Timer;         // Таймер рисования

    sMesh *m_Mesh;                // Исходная сетка
    char *m_VertexPtr;            // Указатель на вершины сетки
    unsigned long m_VertexFVF;    // FVF вершин сетки
    unsigned long m_VertexSize;   // Размер вершины сетки

    // SortNode группирует полигоны в узлы и в случае необходимости
    // разделяет узел на несколько потомков
    void SortNode(sNode *Node,
                 float XPos, float YPos, float ZPos,
                 float Size);

    // AddNode добавляет узел в список рисуемых узлов
    void AddNode(sNode *Node);

    // IsPolygonContained возвращает TRUE если
    // ограничивающий прямоугольник полигона пересекается с
    // указанной кубической областью пространства
    BOOL IsPolygonContained(
        sPolygon *Polygon,
        float XPos, float YPos, float ZPos,
        float Size);

    // CountPolygons возвращает количество полигонов
    // в трехмерном кубе
    unsigned long CountPolygons(
        float XPos, float YPos, float ZPos,
        float Size);
}
```

```

public:
    cNodeTreeMesh(); // Конструктор
    ~cNodeTreeMesh(); // Деструктор

    // Функции для создания и освобождения разделенной
    // на узлы дерева сетки из исходного объекта cMesh с указанием
    // максимального количества полигонов в области, которая больше
    // заданного размера (вызывающего разделение узла)
    BOOL Create(cGraphics *Graphics, cMesh *Mesh,
               int TreeType = OCTREE,
               float MaxSize = 256.0f, long MaxPolygons = 32);
    BOOL Free();

    // Визуализация объекта с использованием текущего преобразования
    // вида и перегруженной дистанции видимости. Также можно указать
    // для использования ранее вычисленную пирамиду видимого
    // пространства, или функция сама вычислит собственную пирамиду.
    BOOL Render(cFrustum *Frustum=NULL, float ZDistance=0.0f);

    // Получение ближайшей высоты выше или ниже точки
    float GetClosestHeight(float XPos, float YPos, float ZPos);

    // Получение высоты выше и ниже точки
    float GetHeightBelow(float XPos, float YPos, float ZPos);
    float GetHeightAbove(float XPos, float YPos, float ZPos);

    // Проверяем, блокирует ли полигон путь от начала до конца
    BOOL CheckIntersect(float XStart, float YStart, float ZStart,
                       float XEnd, float YEnd, float ZEnd,
                       float *Length);
};

```

Комментарии в приведенном выше классе достаточно хорошо описывают, что представляет каждая переменная и что делает каждая функция. В нескольких следующих разделах я опишу каждую функцию более подробно. Сейчас взгляните на таблицу 8.1, где перечислены и описаны все переменные **cNodeTreeMesh**.

Таблица 8.1. Переменные cNodeTreeMesh

<i><b>Переменная</b></i>	<i><b>Описание</b></i>
<b>m_Graphics</b>	Указатель на ранее инициализированный объект <b>cGraphics</b>
<b>m_Frustum</b>	Указатель на класс пирамиды видимого пространства <b>cFrustum</b> , который будет использоваться при операциях визуализации
<b>m_Timer</b>	Текущий кадр (используется при визуализации для предотвращения перерисовок)
<b>m_Size</b>	Размер мира от центра (начала координат) до дальней грани по любой из трех осей (мир является кубом)
<b>m_MaxSize</b>	Максимальный размер узла, который может быть перед разделением (оно происходит, когда узел содержит слишком много полигонов)

Таблица 8.1.(продолжение) Переменные cNodeTreeMesh

Переменная	Описание
<b>m_ParentNode</b>	Голова связанного списка узлов
<b>m_NumGroups</b>	Количество групп материалов
<b>m_Groups</b>	Массив групп материалов
<b>m_NumPolygons</b>	Количество полигонов в исходной сетке
<b>m_MaxPolygons</b>	Хранит максимальное количество полигонов, допустимое для узла (при его превышении происходит разделение узла)
<b>m_PolygonList</b>	Массив структур <b>sPolygon</b> , хранящих информацию о каждом полигоне сетки
<b>m_Mesh</b>	Указатель на исходную сетку (структуру <b>sMesh</b> , используемую графическим ядром)
<b>m_VertexPtr</b>	Используется для доступа к буферу вершин исходной сетки
<b>m_VertexFVF</b>	Дескриптор FVF исходной сетки
<b>m_VertexSize</b>	Хранит размер одной вершины исходной сетки (в байтах)

Продолжайте чтение главы, и я укажу, когда переменные класса вступают в игру, а сейчас пришло время больше узнать о функциях класса **cNodeTreeMesh**.

### **cNodeTreeMesh::Create u cNodeTreeMesh::Free**

Функция **cNodeTreeMesh::Create** — это место где начинается весь процесс. Для создания структуры NodeTree вам необходима исходная сетка с которой мы будем работать, и лучше всего взять ее из объекта **cMesh**. Объект **cMesh** хранит список сеток, содержащихся в одном X-файле, и мы для простоты будем иметь дело только с первой сеткой из списка.

Передав функции **Create** указатель на инициализированный объект **cGraphics**, предварительно загруженный объект **cMesh** и тип дерева, который вы хотите использовать (**QUADTREE** или **OCTREE**), вы инициализируете класс, после чего он готов к визуализации.

Помимо уже упомянутых аргументов вам надо задать максимальное количество полигонов для узла и максимальный размер узла, прежде чем ему потребуется разделение (когда он содержит максимально возможное для узла количество полигонов).

```

BOOL cNodeTreeMesh::Create(cGraphics *Graphics, cMesh *Mesh,
                           int TreeType,
                           float MaxSize, long MaxPolygons)
{
    ID3DXMesh      *LoadMesh;
    unsigned short *IndexPtr;

```



```
unsigned long   *Attributes;
float           MaxX, MaxY, MaxZ;
unsigned long   i;

// Освобождаем предыдущую сетку
Free();

// Проверка ошибок
if((m_Graphics = Graphics) == NULL)
    return FALSE;
if(Mesh == NULL)
    return FALSE;
if(!Mesh->GetParentMesh()->m_NumMaterials)
    return FALSE;

// Получаем сведения о сетке
m_Mesh          = Mesh->GetParentMesh();
LoadMesh        = m_Mesh->m_Mesh;
m_VertexFVF     = LoadMesh->GetFVF();
m_VertexSize    = D3DXGetFVFVertexSize(m_VertexFVF);
m_NumPolygons   = LoadMesh->GetNumFaces();
m_MaxPolygons   = MaxPolygons;

// Создаем список полигонов и групп
m_Polygons      = new sPolygon[m_NumPolygons]();
m_NumGroups     = m_Mesh->m_NumMaterials;
m_Groups        = new sGroup[m_NumGroups]();

// Блокируем буфер индексов и буфер атрибутов
LoadMesh->LockIndexBuffer(D3DLOCK_READONLY, (void**)&IndexPtr);
LoadMesh->LockAttributeBuffer(D3DLOCK_READONLY, &Attributes);
```

Здесь мы видим кое-что новое. **ID3DXMesh** использует буфер вершин; кроме того, **ID3DXMesh** использует буфер индексов! Вы только что читали об этом в предыдущем разделе; D3DX использует буферы вершин и индексов для экономии памяти и ускорения визуализации, поскольку буфер индексов позволяет во время визуализации несколько раз использовать одну и ту же вершину без необходимости несколько раз сохранять координаты вершины в буфере вершин.

Также у объекта **ID3DXMesh** есть нечто, называемое *буфер атрибутов* (*attribute buffer*), представляющий собой массив значений, определяющих какой материал какой полигон использует. Между элементами массива и полигонами существует отношение один к одному. Именно этот буфер атрибутов используется, чтобы включать полигоны в соответствующие им группы материалов.

В последующем коде начинается перебор всех полигонов сетки с извлечением трех вершин, используемых для конструирования каждого полигона. Помимо этого, вы в структуре данных полигона сохраняете материал, применяемый для рисования этого полигона.

```
// Загрузка данных полигонов в структуры
for(i = 0; i < m_NumPolygons; i++) {
    m_Polygons[i].Vertex0 = *IndexPtr++;
    m_Polygons[i].Vertex1 = *IndexPtr++;
    m_Polygons[i].Vertex2 = *IndexPtr++;
```

```

        m_Polygons[i].Group = Attributes[i];
        m_Polygons[i].Timer = 0;
        m_Groups[Attributes[i]].NumPolygons++;
    }

    // Разблокирование буферов
    LoadMesh->UnlockAttributeBuffer();
    LoadMesh->UnlockIndexBuffer();

    // Построение индексных буферов групп вершин
    for(i = 0; i < m_NumGroups; i++) {
        if(m_Groups[i].NumPolygons != 0)
            m_Graphics->GetDeviceCOM()->CreateIndexBuffer(
                m_Groups[i].NumPolygons * 3 * sizeof(unsigned short),
                D3DUSAGE_WRITEONLY, D3DFMT_INDEX16, D3DPOOL_MANAGED,
                &m_Groups[i].IndexBuffer, NULL);
    }

```

В показанном выше коде массив индексов и буфер атрибутов сетки заблокированы. После этого вы просто строите список данных полигонов (какие вершины образуют полигон и какая группа материалов используется). Завершив построение списка данных полигонов, вы разблокируете буферы индексов и атрибутов и формируете фактические группы материалов.

Каждая группа материалов содержит объект **IDirectIndexBuffer9** для хранения индексов вершин, которые надо рисовать. Чтобы создать этот буфер индексов вы используете функцию **CreateIndexBuffer**, во многом похожую на функцию **CreateVertexBuffer**, о которой вы читали в главе 2. Единственное различие в том, что буфер индексов использует для значений индексов переменные типа **short** (2 байта), и выделяет по три индекса на полигон. Обратите внимание, что нужно выделить достаточно индексов для каждого полигона, относящегося к каждой группе материалов.

```

// Получаем размер ограничивающего куба
MaxX = (float)max(fabs(Mesh->GetParentMesh()->m_Min.x),
                  fabs(Mesh->GetParentMesh()->m_Max.x));
MaxY = (float)max(fabs(Mesh->GetParentMesh()->m_Min.y),
                  fabs(Mesh->GetParentMesh()->m_Max.y));
MaxZ = (float)max(fabs(Mesh->GetParentMesh()->m_Min.z),
                  fabs(Mesh->GetParentMesh()->m_Max.z));
m_Size = max(MaxX, max(MaxY, MaxZ)) * 2.0f;
m_MaxSize = MaxSize;

// Создаем родительский узел
m_ParentNode = new sNode();

```

Здесь вычисляется точка наиболее удаленная от центра исходной сетки. Для упрощения вычислений мир считается идеальным кубом, поэтому мы используем расстояние от центра сетки до внешнего края в качестве размера мирового ограничивающего параллелепипеда. После создания родительского узла (**m\_ParentNode**) вы блокируете буфер вершин сетки чтобы прочитать координаты вершин. И, наконец, узлы сортируются путем вызова функции **SortNode**.

```
// Сортировка полигонов в узлах
LoadMesh->LockVertexBuffer(D3DLOCK_READONLY,
                           (void**)&m_VertexPtr);
SortNode(m_ParentNode, 0.0f, 0.0f, 0.0f, m_Size);
LoadMesh->UnlockVertexBuffer();

m_Timer = 0; // Сброс таймера

return TRUE;
}
```

После сортировки буфер вершин сетки разблокируется и функция возвращает **TRUE**, сообщая об успешном завершении. Теперь представление сетки в виде узлов дерева готово к использованию. Завершив работу с этим представлением вы вызываете **cNodeTreeMesh::Free** для освобождения ресурсов сетки:

```
BOOL cNodeTreeMesh::Free()
{
    delete m_ParentNode; // Удаляем родительский и дочерние узлы
    m_ParentNode = NULL;

    m_NumPolygons = 0; // Больше нет полигонов
    delete [] m_PolygonList; // Удаляем массив полигонов
    m_PolygonList = NULL;

    m_NumGroups = 0; // Больше нет групп материалов
    delete [] m_Groups; // Удаляем группы материалов
    m_Groups = NULL;

    m_Graphics = NULL;

    return TRUE;
}
```

### **cNodeTreeMesh::SortNode**

**cNodeTreeMesh::SortNode** — это рекурсивная функция (вызывающая сама себя), которая подсчитывает количество полигонов, находящихся в узле трехмерного пространства и решает надо ли разделить узел на четыре или восемь дочерних узлов (в зависимости от типа дерева). После того, как узлы соответствующим образом разделены, функция **SortNode** строит список полигонов, находящихся в узле трехмерного пространства.

Код функции **SortNode** начинается с проверки правильности переданных ей аргументов. Затем функция сохраняет координаты узла и начинает деление узлов.

---

#### **ПРИМЕЧАНИЕ**

Если вы используете квадродерево, в операциях отсечения и деления надо игнорировать высоту. Обратите внимание, что в зависимости от режима в коде класса **cNodeTreeMesh** значения высоты игнорируются или обрабатываются по-другому.

---

```

void cNodeTreeMesh::SortNode(sNode *Node,
                             float XPos, float YPos, float ZPos,
                             float Size)
{
    unsigned long i, Num;
    float        XOff, YOff, ZOff;

    // Проверка ошибок
    if(Node == NULL)
        return;

    // Сохраняем координаты и размер узла
    Node->XPos = XPos;
    Node->YPos = (m_TreeType==QUADTREE)?0.0f:YPos;
    Node->ZPos = ZPos;
    Node->Size = Size;

    // Смотрим, есть ли полигоны в узле
    if(!(Num = CountPolygons(XPos, YPos, ZPos, Size)))
        return;

    // Разделяем узел, если его размер больше максимума
    // и в нем слишком много полигонов
    if(Size > m_MaxSize && Num > m_MaxPolygons) {
        for(i=0; i < (unsigned long)((m_TreeType==QUADTREE)?4:8); i++) {
            XOff = (((i % 2) < 1) ? -1.0f : 1.0f) * (Size / 4.0f);
            ZOff = (((i % 4) < 2) ? -1.0f : 1.0f) * (Size / 4.0f);
            YOff = (((i % 8) < 4) ? -1.0f : 1.0f) * (Size / 4.0f);

            // Смотрим, есть ли полигоны в ограничивающем
            // параллелепипеде нового узла
            if(CountPolygons(XPos + XOff, YPos + YOff, ZPos + ZOff,
                             Size/2.0f)) {

                // Создаем новый дочерний узел
                Node->Nodes[i] = new sNode();

                // Сортируем полигоны с новым дочерним узлом
                SortNode(Node->Nodes[i], XPos+XOff, YPos+YOff,
                          ZPos+ZOff, Size/2.0f);
            }
        }
        return;
    }
}

```

К этому моменту все полигоны, находящиеся в ограничивающем параллелепипеде узла посчитаны. Если полигонов слишком много и размер ограничивающего параллелепипеда узла достаточно большой, узел разделяется. Полученные в результате деления узлы проходят тот же самый процесс. В процессе пустые узлы исключаются из последующей обработки, а узлы с полигонами отправляются назад функции **SortNode** (это и есть рекурсия).

```

// Выделяем место для списка указателей на полигоны
Node->NumPolygons = Num;
Node->PolygonList = new unsigned long[Num];

// Сканируем список полигонов, сохраняя указатели и назначая их
Num = 0;

```

```

for(i = 0; i < m_NumPolygons; i++) {
    // Добавляем полигон к списку, если он находится
    // в заданной области пространства
    if(IsPolygonContained(&m_PolygonList[i],
                        XPos, YPos, ZPos, Size) == TRUE)
        Node->PolygonList[Num++] = i;
}
}

```

Последний фрагмент кода **SortNode** выделяет массив индексов, указывающих на массив **m\_Polygons**. Поскольку массив **m\_Polygons** содержит данные всех полигонов, имеет смысл сократить объем используемой памяти, сохраняя в массиве **PolygonList** индекс элемента массива **m\_Polygons**. (Индекс 0 в **PolygonList** указывает на данные полигона в **m\_Polygons[0]**, индекс 1 в **PolygonList** указывает на данные полигона в **m\_Polygons[1]** и т.д.)

Когда приходит время визуализации, этот массив указателей используется для получения информации о каждом полигоне, включая индексы вершин и используемую группу материалов. Рисуемые полигоны позднее добавляются к буферу индексов соответствующей группы материалов с помощью функции **AddNode**.

### ***cNodeTreeMesh::IsPolygonContained u cNodeTreeMesh::CountPolygons***

Этот дуэт функций вы используете для того, чтобы определить, находится ли полигон в области трехмерного пространства, заданной ограничивающим параллелепипедом, и подсчитать общее количество полигонов, находящихся в заданной области.

```

BOOL cNodeTreeMesh::IsPolygonContained(sPolygon *Polygon,
                                       float XPos, float YPos, float ZPos,
                                       float Size)
{
    float XMin, XMax, YMin, YMax, ZMin, ZMax;
    sVertex *Vertex[3];

    // Получаем вершины полигона
    Vertex[0] = (sVertex*)&m_VertexPtr[m_VertexSize *
                                       Polygon->Vertex[0]];
    Vertex[1] = (sVertex*)&m_VertexPtr[m_VertexSize *
                                       Polygon->Vertex[1]];
    Vertex[2] = (sVertex*)&m_VertexPtr[m_VertexSize *
                                       Polygon->Vertex[2]];

    // Проверяем ось X заданной области пространства
    XMin = min(Vertex[0]->x, min(Vertex[1]->x, Vertex[2]->x));
    XMax = max(Vertex[0]->x, max(Vertex[1]->x, Vertex[2]->x));
    if(XMax < (XPos - Size / 2.0f))
        return FALSE;
    if(XMin > (XPos + Size / 2.0f))
        return FALSE;
}

```

В показанном выше коде (а также и в показанном ниже) определяется вершина, наиболее удаленная по оси X. Если полигон слишком далеко

сдвинут влево или вправо от проверяемого ограничивающего параллелепипеда, то функция возвращает **FALSE**. То же самое выполняется и для двух других осей.

```
// Проверяем ось Y заданной области пространства (для октодерева)
if(m_TreeType == OCTREE) {
    YMin = min(Vertex[0]->y, min(Vertex[1]->y, Vertex[2]->y));
    YMax = max(Vertex[0]->y, max(Vertex[1]->y, Vertex[2]->y));
    if(YMax < (YPos - Size / 2.0f))
        return FALSE;
    if(YMin > (YPos + Size / 2.0f))
        return FALSE;
}

// Проверяем ось Z заданной области пространства
ZMin = min(Vertex[0]->z, min(Vertex[1]->z, Vertex[2]->z));
ZMax = max(Vertex[0]->z, max(Vertex[1]->z, Vertex[2]->z));
if(ZMax < (ZPos - Size / 2.0f))
    return FALSE;
if(ZMin > (ZPos + Size / 2.0f))
    return FALSE;

return TRUE;
}

unsigned long cNodeTreeMesh::CountPolygons(
    float XPos, float YPos, float ZPos, float Size)
{
    unsigned long i, Num;

    // Возвращаемся, если нет полигонов для обработки
    if(!m_NumPolygons)
        return 0;

    // Перебираем каждый полигон и считаем те,
    // которые находятся в заданной области пространства
    Num = 0;
    for(i = 0; i < m_NumPolygons; i++) {
        if(IsPolygonContained(&m_PolygonList[i], XPos, YPos, ZPos,
            Size) == TRUE)
            Num++;
    }

    return Num;
}
```

Как видите, **CountPolygons** в цикле перебирает все полигоны сетки и проверяет, находятся ли они внутри ограничивающего куба. Попавшие в заданное пространство полигоны подсчитываются и финальное значение счетчика возвращается вам. Функция используется чтобы определить сколько полигонов находится внутри узла при разделении и сортировке узлов.

### ***cNodeTreeMesh::AddNode***

Вы используете функцию **cNodeTreeMesh::AddNode** совместно с функцией **Render**. **AddNode** выполняет проверку видимого пространства

для всех узлов и рекурсивно проверяет их дочерние узлы. При первом вызове функции **AddNode** ей передается переменная **m\_ParentNode** для начала процесса с корневого узла.

```
void cNodeTreeMesh::AddNode(sNode *Node)
{
    unsigned long i, Group;
    sPolygon *Polygon;
    short Num;

    if(Node == NULL)
        return;

    // Проверка пирамиды видимого пространства
    // в зависимости от типа дерева
    BOOL CompletelyContained = FALSE;
    if(m_TreeType == QUADTREE) {
        if(m_Frustum->CheckRectangle(
            Node->XPos, 0.0f, Node->ZPos,
            Node->Size/2.0f, m_Size/2.0f,
            Node->Size / 2.0f,
            &CompletelyContained) == FALSE)
            return;
    } else {
        if(m_Frustum->CheckRectangle(
            Node->XPos, Node->YPos, Node->ZPos,
            Node->Size/2.0f, Node->Size/2.0f,
            Node->Size/2.0f,
            &CompletelyContained) == FALSE)
            return;
    }
}
```

И снова вы видите проверку пирамиды видимого пространства, которую я упоминал ранее. Здесь режимы квадродерева и октодерева различаются. Для квадродерева, являющегося двухмерной структурой, проверяются только два измерения (Y для двухмерного представления всегда отбрасывается). В случае октодерева узел может находиться в любом месте трехмерного пространства, поэтому как минимум одна точка должна быть видима.

---

<b>ПРИМЕЧАНИЕ</b>	Если узел находится вне пирамиды видимого пространства, все его потомки отбрасываются. Также, если узел полностью находится в пирамиде видимого пространства, нет необходимости проверять его потомков, поскольку они тоже будут внутри пирамиды. Так можно ускорить работу с деревом!
-------------------	--

---

Теперь функция **AddNode** решает, должны ли быть добавлены дочерние узлы (в том случае, если они содержат полигоны). Добавление узлов заканчивается, когда у текущего узла нет потомков. В этом случае обрабатывается следующий родительский узел и так до тех пор, пока не будут обработаны все узлы. (Обратите внимание, что если узел полностью находится внутри пирамиды видимого пространства, проверять его дочерние узлы не надо.)

```

if(CompletelyContained == FALSE) {

    // Сканируем другие узлы
    Num = 0;
    for(i=0; i < (unsigned long)((m_TreeType==QUADTREE)?4:8); i++) {
        if(Node->Nodes[i] != NULL) {
            Num++;
            AddNode(Node->Nodes[i]);
        }
    }
    // Не надо продолжать, если есть другие узлы
    if(Num)
        return;
}

```

Теперь функция **AddNode** проверяет, содержит ли рассматриваемый узел какие-нибудь полигоны. Если в узле есть полигоны, каждый из них проверяется видим ли он (полигоны у которых альфа-значение материала равно 0.0, считаются невидимыми).

**AddNode** добавляет видимые полигоны в буфер индексов соответствующей группы материалов и сохраняет количество рисуемых в узле полигонов. В каждом кадре количество полигонов, визуализируемых в каждой группе материалов, сбрасывается функцией **Render**. Следующий код увеличивает количество рисуемых полигонов:

---

<b>ПРИМЕЧАНИЕ</b>	Каждому полигону назначена переменная таймера, поскольку если полигон уже нарисован в текущем кадре, его не надо перерисовывать (при обработке других узлов на которые распространяется полигон). Поэтому <b>AddNode</b> проверяет переменную таймера каждого полигона, прежде чем добавить его к буферу индексов группы материалов.
-------------------	--

---

```

// Добавляем содержащиеся полигоны (если есть)
if(Node->NumPolygons != 0) {
    for(i = 0; i < Node->NumPolygons; i++) {
        // Получаем указатель на полигон
        Polygon = &m_Polygons[Node->PolygonList[i]];

        // Рисуем, только если уже
        // не был нарисован в этом кадре
        if(Polygon->Timer != m_Timer) {
            // Отмечаем полигон как обработанный в этом кадре
            Polygon->Timer = m_Timer;

            // Получаем группу материалов полигона
            Group = Polygon->Group;

            // Проверяем правильность указания группы
            // и что материал непрозрачный
            if(Group < m_NumGroups &&
                m_Mesh->m_Materials[Group].Diffuse.a != 0.0f) {
                // Копируем индексы в буфер индексов
                *m_Groups[Group].IndexPtr++ = Polygon->Vertex0;
                *m_Groups[Group].IndexPtr++ = Polygon->Vertex1;
                *m_Groups[Group].IndexPtr++ = Polygon->Vertex2;
            }
        }
    }
}

```



```

        // Увеличиваем счетчик рисуемых полигонов в группе
        m_Groups[Group].NumPolygonsToDraw++;
    }
}
}
}
}

```

После завершения функции **AddNode** вы получаете полный набор буферов индексов групп материалов, готовый к визуализации.

### ***cNodeTreeMesh::Render***

Вы дошли до конца класса **cNodeTreeMesh**, и разве может быть лучшее место для размещения функции, вознаграждающей вас визуализацией полигонов? Функции **cNodeTreeMesh::Render** можно передать необязательную пирамиду видимого пространства, которая будет использоваться вместо внутренней пирамиды видимого пространства.

Кроме того, поскольку функция визуализации в работе использует пирамиду видимого пространства, необходим способ переопределять глубину поля зрения. Это поможет, если вы захотите рисовать только те полигоны сетки, которые находятся не далее указанного расстояния от точки просмотра. Например, если протяженность пирамиды видимого пространства составляет 20 000 единиц, а вы хотите визуализировать только полигоны, находящиеся в пределах 5 000 единиц, задайте 5 000 для **ZDistance**.

Функция **Render** начинается с вычисления пирамиды видимого пространства и блокировки буфера вершин:

```

BOOL cNodeTreeMesh::Render(cFrustum *Frustum, float ZDistance)
{
    D3DXMATRIX Matrix;        // Матрица, используемая для вычислений
    cFrustum ViewFrustum;      // Локальная пирамида видимого пространства

    IDirect3DVertexBuffer9 *pVB = NULL;

    unsigned long i;

    // Проверка ошибок
    if(m_Graphics==NULL || m_ParentNode==NULL || !m_NumPolygons)
        return FALSE;

    // Конструируем пирамиду видимого пространства
    // (если она не передана в параметре)
    if((m_Frustum = Frustum) == NULL) {
        ViewFrustum.Construct(m_Graphics, ZDistance);
        m_Frustum = &ViewFrustum;
    }

    // Делаем матрицу мирового преобразования единичной,
    // чтобы сетка уровня визуализировалась вокруг начала координат
    // как была разработана
    D3DXMatrixIdentity(&Matrix);
    m_Graphics->GetDeviceCOM()->SetTransform(D3DTS_WORLD,&Matrix);
}

```

```
// Блокируем буфер индексов группы
for(i = 0; i < m_NumGroups; i++) {
    if(m_Groups[i].NumPolygons) {
        m_Groups[i].IndexBuffer->Lock(
            0, m_Groups[i].NumPolygons * 3 * sizeof(unsigned short),
            (void**) &m_Groups[i].IndexPtr, 0);
    }
    m_Groups[i].NumPolygonsToDraw = 0;
}
```

Сейчас буферы индексов групп материалов заблокированы и готовы принимать информацию об индексах от функции **AddNode**. В каждой группе материалов обнуляется количество рисуемых полигонов, и в следующих строках кода все видимые полигоны (через узлы) будут добавлены к буферам индексов групп материалов.

Теперь увеличивается значение переменной **m\_Timer** (чтобы увеличить счетчик кадров и гарантировать, что полигоны не будут рисоваться по несколько раз в одном и том же кадре) и вызывается функция **AddNode** для родительского узла.

```
// Увеличиваем счетчик кадров
m_Timer++;

// Добавляем рисуемые полигоны в списки групп
AddNode(m_ParentNode);
```

Вы готовы визуализировать видимые полигоны. Для этого вам необходим доступ к буферу вершин исходной сетки; здесь придет на помощь вызов **ID3DXMesh::GetVertexBuffer**. Затем вы устанавливаете соответствующий потоковый источник данных, шейдер FVF, и в цикле перебираете каждую группу материалов, устанавливая каждый материал и текстуру и визуализируя полигоны с помощью вызова **IDirect3DDevice9::DrawIndexedPrimitive**. Разблокируйте буфер вершин сетки и сделайте все необходимое!

```
// Получаем указатель на буфер вершин
m_Mesh->m_Mesh->GetVertexBuffer(&pVB);

// Устанавливаем вершинный шейдер и источник данных
m_Graphics->GetDeviceCOM()->SetStreamSource(0, pVB,
                                             0, m_VertexSize);
m_Graphics->GetDeviceCOM()->SetFVF(m_VertexFVF);

// Разблокируем буфер вершин и рисуем
for(i = 0; i < m_NumGroups; i++) {
    if(m_Groups[i].NumPolygons)
        m_Groups[i].IndexBuffer->Unlock();
    if(m_Groups[i].NumPolygonsToDraw) {
        m_Graphics->GetDeviceCOM()->SetMaterial(
            &m_Mesh->m_Materials[i]);
        m_Graphics->GetDeviceCOM()->SetTexture(
            0, m_Mesh->m_Textures[i]);
        m_Graphics->GetDeviceCOM()->SetIndices(
            m_Groups[i].IndexBuffer);
    }
}
```

```
m_Graphics->GetDeviceCOM()->DrawIndexedPrimitive(
    D3DPT_TRIANGLELIST, 0, 0,
    m_Mesh->m_Mesh->GetNumVertices(), 0,
    m_Groups[i].NumPolygonsToDraw);
}
}

// Освобождаем буфер вершин
if(pVB != NULL)
    pVB->Release();

return TRUE;
}
```

Вы можете недоумевать, как в целом работает эта схема визуализации с буфером индексов. В Direct3D вы должны указать, какой буфер индексов и какой буфер вершин используются для визуализации. Это делается с помощью функций **SetIndices** и **SetStreamSource**.

Когда выполняется сама визуализация, вы, вместо вызова функции **DrawPrimitive**, о которой читали в главе 2, обращаетесь к функции **DrawIndexedPrimitive**. Эта функция использует предоставленные вами буферы вершин и индексов для рисования полигонов заданного типа (списков треугольников, полос, вееров и т.д.). Параметры **DrawIndexedPrimitive** те же самые, что у **DrawPrimitive**, так что здесь проблем возникнуть не должно.

## Использование cNodeTreeMesh

Если в своем проекте вы используете графическое ядро, для применения класса **cNodeTreeMesh** достаточно включить его в проект, предоставить исходную сетку, с которой он будет работать и визуализировать ее! С другой стороны, если вы не хотите использовать графическое ядро, потребуется некоторая переработка. В приведенном ниже коде вы увидите фрагменты, где используются классы графического ядра (я постарался свести их число к минимуму), и сейчас у вас достаточно знаний, чтобы приспособить движок NodeTree к вашему собственному графическому ядру.

Демонстрационная программа NodeTree есть на прилагаемом к книге CD-ROM. Вы найдете ее в каталоге BookCode\Chap08\NodeTree. А сейчас взгляните на код, показывающий как быстро построить дерево узлов и визуализировать сетку:

```
// Graphics = ранее инициализированный объект cGraphics
cMesh          Mesh;
cNodeTreeMesh  NodeTreeMesh;
cCamera        Camera;
cFrustum       Frustum;

// Загружаем сетку с диска
Mesh.Load(&Graphics, "mesh.x");
NodeTreeMesh.Create(&Graphics, &Mesh, OCTREE);
```

---

```
// Устанавливаем позицию камеры и
// создаем пирамиду видимого пространства
Camera.Point(0.0f, 100.0f, -200.0f, 0.0f, 0.0f, 0.0f);
Graphics.SetCamera(&Camera);
Frustum.Construct(&Graphics);

// Начинаем сцену, визуализируем сетку,
// завершаем сцену и отображаем ее

// Визуализируем все
Graphics.Clear(D3DCOLOR_RGBA(0,0,0,0));
if(Graphics.BeginScene() == TRUE) {
    NodeTreeMesh.Render(&Frustum);
    Graphics.EndScene();
}
Graphics.Display();

// Освобождаем все
NodeTreeMesh.Free();
Mesh.Free();
```

**ПРИМЕЧАНИЕ**

Поэкспериментировав с движком NodeTree вы заметите, что отдельные части вашего уровня то появляются, то исчезают. Это вызвано тем, что пирамида видимого пространства отсекает большие секции вашего уровня целиком.

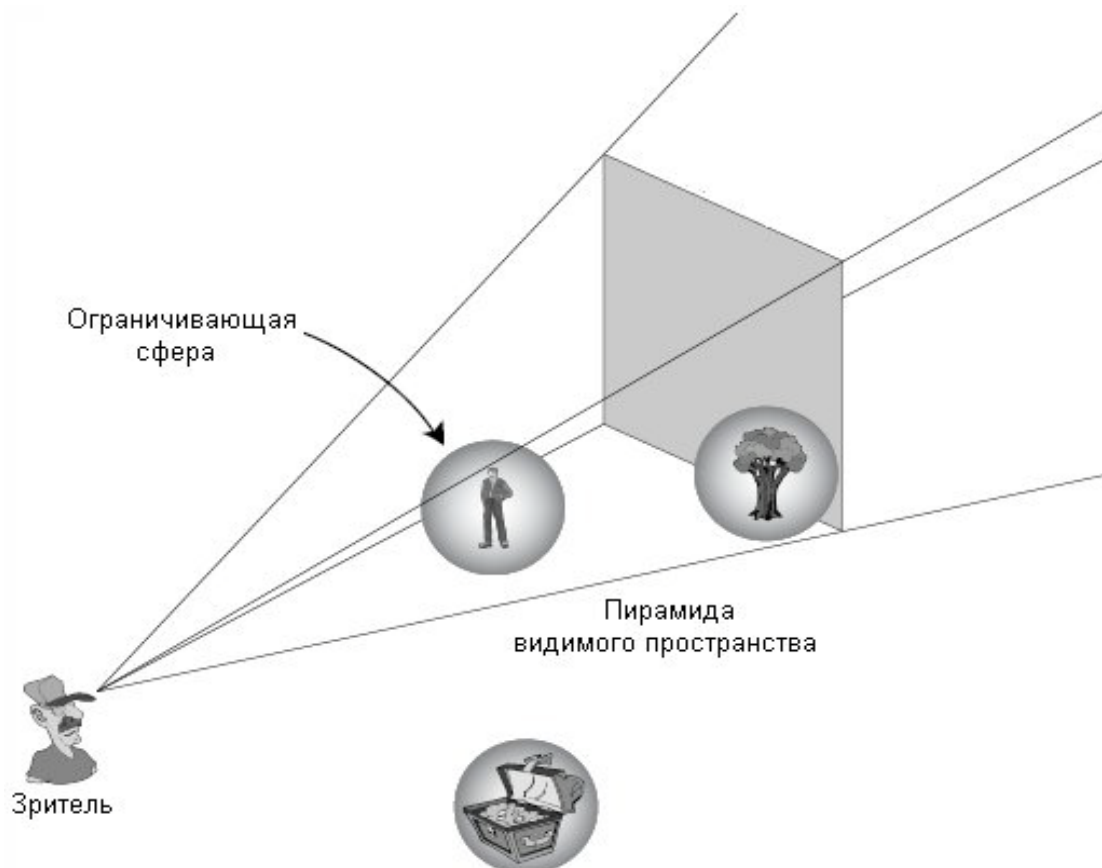
---

## Добавление к миру трехмерных объектов

Трехмерный мир не будет завершенным без объектов. Поскольку вы используете трехмерный мир, в качестве игровых объектов хорошо использовать трехмерные сетки; однако, как вы видели в разделе «Сетки в качестве уровней», тупое рисование тысяч объектов (без какого-либо отбрасывания) вызовет существенные задержки в конвейере визуализации графики.

Здесь нам снова потребуется использовать пирамиду видимого пространства для быстрого определения того, какие объекты видимы, а какие нет. Чтобы определить видимость трехмерного объекта вы заключаете его в невидимую сферу, называемую *ограничивающей сферой (bounding sphere)*, которая будет использоваться совместно с функцией **CheckSphere** класса пирамиды видимого пространства (смотрите раздел «Класс cFrustum» ранее в этой главе).

На рис. 8.7 показано использование ограничивающих сфер и пирамиды видимого пространства. На иллюстрации вы видите сцену с тремя объектами и пирамидой видимого пространства. У каждой сетки есть невидимая ограничивающая сфера, внутри которой она находится. Вспомните, что сфера считается находящейся в поле зрения, если она лежит перед всеми шестью плоскостями, образующими пирамиду видимого пространства.



*Рис. 8.7. Вы можете проверить три объекта в сцене относительно пирамиды видимого пространства*

На рис. 8.7 только два объекта находятся в пирамиде видимого пространства и видны зрителю. Один объект полностью находится вне пирамиды видимого пространства. Вы захотите иметь возможность рисовать только видимые объекты и пропускать те, которые расположены вне пирамиды. Для этого вы должны вычислить ограничивающую сферу каждого объекта и затем проверить ее, чтобы убедиться, находится ли она внутри пирамиды.

## Вычисление ограничивающей сферы

Если вы не используете класс сеток графического ядра, вам необходимо определить ограничивающую сферу сетки. Используя D3DX, вы можете обратиться к функции **D3DXComputeBoundingSphere** (именно она используется в графическом ядре):

```
HRESULT D3DXComputeBoundingSphere(
    PVOID          pvPointsFVF, // Буфер, содержащий вершины
    DWORD          NumVertices, // Количество вершин в буфере
    DWORD          FVF,         // Дескриптор FVF
    D3DXVECTOR3    *pCenter,    // &D3DXVECTOR2(0.0f, 0.0f, 0.0f)
    FLOAT          *pRadius);   // Указатель на переменную радиуса
```

Эта функция просто сканирует буфер вершин, запоминая какая вершина больше всего удалена от центра. Расстояние от центра до этой вершины и является радиусом сферы. Перед вызовом **D3DXComputeBoun-**

**dingSphere** вам надо заблокировать буфер вершин с помощью следующего вызова:

```
HRESULT ID3DXMesh::LockVertexBuffer(
    DWORD Flags,      // Флаги блокировки. Используйте D3DLOCK_READONLY
    BYTE **ppData); // Указатель на данные буфера вершин
```

---

**ПРИМЕЧАНИЕ** **ID3DXMesh** и **ID3DXSkinMesh** используют для блокировки буфера вершин одну и ту же функцию, так что вызов для **ID3DXMesh** подходит и для **ID3DXSkinMesh**.

---

Когда сетка загружена, все готово к вычислению ее ограничивающей сферы, как показано ниже:

```
// pMesh = ранее загруженный объект ID3DXMesh
float Radius; // Радиус объекта
BYTE **Ptr; // Указатель на буфер вершин

// Блокируем буфер вершин
if(SUCCEEDED(pMesh->LockVertexBuffer(D3DLOCK_READONLY,
    (BYTE**) &Ptr))) {

    // Вычисляем радиус ограничивающей сферы
    D3DXComputeBoundingSphere((void*)Ptr,
        pMesh->GetNumVertices(), pMesh->GetFVF(),
        &D3DXVECTOR3(0.0f, 0.0f, 0.0f), &Radius);

    // Разблокируем буфер вершин
    pMesh->UnlockVertexBuffer();
}
```

Обратите внимание, что в вызове **D3DXComputeBoundingSphere** количество вершин и дескриптор FVF запрашиваются непосредственно у **ID3DXMesh**. За вызовом **ID3DXMesh::LockVertexBuffer** следует вызов **ID3DXMesh::UnlockVertexBuffer**. Вы должны разблокировать буфер вершин, закончив работу с ним, иначе последующие вызовы блокировки или визуализации буфера приведут к сбою.

## Ограничивающие сферы и пирамида видимого пространства

Сейчас, когда у вас сконструирована пирамида видимого пространства и есть ограничивающая сфера, можно вызвать функцию **cFrustum::CheckSphere**, чтобы проверить видим ли объект:

```
// cFrustum *Frustum = ранее созданная пирамида видимого пространства
// Radius = ранее вычисленный радиус ограничивающей сферы
// XPos, YPos, ZPos = Мировые координаты объекта
if(Frustum->CheckSphere(XPos, YPos, ZPos, Radius) == TRUE) {

    // Объект внутри пирамиды, переходим к его визуализации

} else {

    // Объект вне пирамиды, пропускаем визуализацию

}
```

## Обнаружение столкновений сеток

Даже использование для представления уровней простых сеток может вызвать большие проблемы, например, в том случае, когда трехмерные объекты сталкиваются с другими объектами мира, или когда пользователь щелкает мышью по сетке (столкновение мыши с сеткой). Например, как вы узнаете, когда ваш игрок или другие свободно передвигающиеся персонажи столкнутся со стенами, или когда сетки столкнутся между собой? Обнаружение таких проблем называется *обнаружением столкновений* (*collision detection*). В этом разделе мы узнаем как определить, что сетка сталкивается с другой сеткой и как определить, что на эту сетку (с помощью мыши) указал пользователь.

### Столкновение с миром

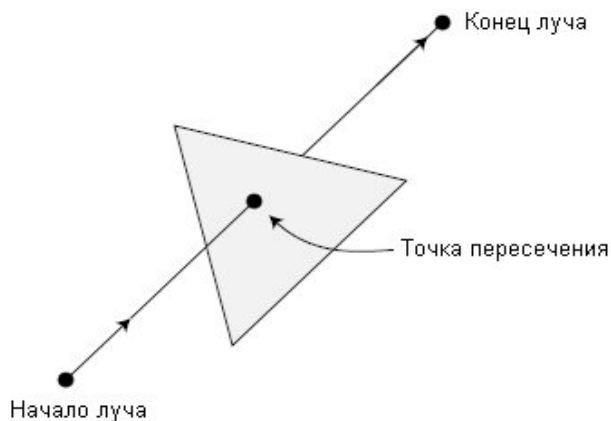
Не знаю, как насчет ваших игр, но у меня большинство персонажей не являются супергероями и не могут проходить сквозь стены! Поэтому трехмерный движок должен знать, когда блокировать перемещение персонажа, если он сталкивается с такими препятствиями, как стена.

#### Испускание лучей

Чтобы проверить, блокирует ли полигон путь от одной точки до другой, вы создаете воображаемый луч, идущий по этим точкам и проверяете, пересекает ли он плоскость. Помните плоскости? Я говорил о них в разделе «Плоскости и отсечение». Полигон — это просто плоскость с точно заданными размерами. Создав представляющую полигон плоскость, вы можете использовать алгебру для обнаружения пересечений (рис. 8.7).

В разделе «Проверка видимости с плоскостями», вы узнали о скалярном произведении и вычислении расстояния от точки до плоскости. Используя те же самые вычисления, вы можете определить пересекает ли точка, движущаяся в трехмерном пространстве плоскость, и если да, то где. Движение точки создает линию, представляющую путь объекта.

Все это показано на рис. 8.8. На иллюстрации вы видите полигон и линию. Линия представляет луч, идущий от начальной точки до конечной. На полпути луч пересекает полигон.



**Рис. 8.8.** Полигон блокирует путь лучу

Помните, что плоскость бесконечна, а значит, луч обязательно пересечет ее, если только он не параллелен плоскости. Поэтому необходима возможность определить, происходит ли пересечение внутри границ полигона, а это сделать несколько труднее.

Здесь на помощь приходит **D3DX**! Единственная функция выполняет проверку пересечения, гарантирует, что пересечение луча с плоскостью происходит внутри полигона, сообщает точные координаты точки пересечения и в качестве подарка возвращает расстояние от начала луча до точки пересечения. Поговорим об этой полезной функции! Итак, что это за функция? Это **D3DXIntersect**, прототип которой выглядит так:

```
HRESULT D3DXIntersect(
    LPD3DXBASEMESH    pMesh,           // Сетка для проверки пересечения
    CONST D3DXVECTOR3 *pRayPos,        // Начало луча
    CONST D3DXVECTOR3 *pRayDir,        // Направление луча
    BOOL               *pHit,           // Флаг наличия пересечения
    DWORD              *pFaceIndex,     // Грань, которую пересекает луч
    FLOAT              *pU,
    FLOAT              *pV,
    FLOAT              *pDist,          // Расстояние до точки
                                         // пересечения
    LPD3DXBUFFER       *ppAllHits,     // Установите NULL
    DWORD              *pCountOfHits); // Установите NULL
```

Сразу видно, что **D3DXIntersect** работает с сетками (**ID3DXBaseMesh**), и это очень хорошо, поскольку вы работаете с объектами сеток. Затем вы видите, что надо указать начальную точку луча (в **pRayPos**). Для **pRayPos** (и **pRayDir**) вы можете использовать макросы **D3DXVECTOR3**.

Аргумент **pRayDir** представляет вектор направления, похожий на вектор нормали. Например, чтобы луч был направлен вверх, вы задаете значение  $-1.0$  для компоненты **Y**. Из остальных аргументов мы будем иметь дело только с **pDist**. Это указатель на переменную типа **FLOAT**, в которую будет записано расстояние от начала луча до точки пересечения.

## Блокировка пути

Теперь, узнав о делающей все функции проверки пересечения, может протестируем ее? Вот небольшой пример, показывающий функцию **D3DXIntersect** в действии. Приведенная ниже функция получает указатель на сетку (представляющую ваш уровень) для проверки пересечений, плюс начальную и конечную точки линии, пересечение с которой проверяется. В ответ вы получите расстояние до места пересечения и точные координаты точки пересечения.

```
BOOL CheckIntersection(ID3DXMesh *Mesh,
                      float XStart, float YStart, float ZStart,
                      float XEnd, float YEnd, float ZEnd,
                      float *Distance)
{
    BOOL Hit; // Флаг наличия пересечения
```



```
float u, v, Dist; // Рабочие переменные и
                  // расстояние до пересечения
float XDiff, YDiff, ZDiff, Size; // Разность и размер
DWORD FaceIndex; // Пересеченная грань
D3DXVECTOR3 vecDir; // Вектор направления для луча

// Вычисляем разность между начальной и конечной точками
XDiff = XEnd - XStart;
YDiff = YEnd - YStart;
ZDiff = ZEnd - ZStart;

// Вычисляем вектор направления
D3DXVec3Normalize(&vecDir, &D3DXVECTOR3(XDiff, YDiff, ZDiff));

// Выполняем проверку пересечения
D3DXIntersect(Mesh,
               &D3DXVECTOR3(XStart, YStart, ZStart), &vecDir,
               &Hit, &FaceIndex, &u, &v, &Dist, NULL, NULL);

// Если обнаружено пересечение, смотрим, находится ли оно
// в пределах линии (расстояние до точки пересечения
// должно быть меньше, чем длина линии)
if(Hit == TRUE) {

    // Получаем длину луча
    Size = (float)sqrt(XDiff*XDiff+YDiff*YDiff+ZDiff*ZDiff);

    // Луч не пересекается с полигоном
    if(Dist > Size)
        Hit = FALSE;
    else {
        // Луч пересекает полигон, сохраняем
        // расстояние до точки пересечения
        if(Length != NULL)
            *Length = Dist;
    }
}
// Возвращаем TRUE если пересечение произошло
// и FALSE если нет
return Hit;
}
```

### **Перемещение вверх и вниз**

Одна из дополнительных выгод использования обнаружения столкновений на уровне полигонов заключается в том, что вы можете заставить объект следовать изменениям высоты находящихся под ним полигонов. Другими словами, мы можем воссоздать хождение по поверхности полигонов! Посмотрите, как это здорово! Представьте себе возможность рисовать уровни в программе трехмерного моделирования и не беспокоиться об определении тех областей, по которым могут ходить игроки — сами полигоны определяют это! Это делает работу с сетками квадродеревьев и октодеревьев еще проще.

Для выполнения проверки пересечения с нижележащей поверхностью в движок NodeTree добавлены три функции. Это показанные ниже функции **GetClosestHeight**, **GetHeightAbove** и **GetHeightBelow**:

```

float GetClosestHeight(ID3DXMesh *Mesh,
                      float XPos, float YPos, float ZPos)
{
    float YAbove, YBelow;

    // Получаем высоту над и под проверяемой точкой
    YAbove = GetHeightAbove(Mesh, XPos, YPos, ZPos);
    YBelow = GetHeightBelow(Mesh, XPos, YPos, ZPos);

    // Смотрим, какая из высот ближе к проверяемой точке
    // и возвращаем это значение
    if(fabs(YAbove-YPos) < fabs(YBelow-YPos))
        return YAbove;    // Ближе высота над, возвращаем ее

    return YBelow;        // Ближе высота под, возвращаем ее
}

float GetHeightBelow(ID3DXMesh *Mesh,
                    float XPos, float YPos, float ZPos)
{
    BOOL Hit;           // Флаг касания полигона
    float u, v, Dist;    // Рабочие переменные и расстояние
                        // до точки пересечения
    DWORD FaceIndex;    // Грань, которую пересекает луч

    // Проводим тест пересечения для сетки
    D3DXIntersect(Mesh,
                  &D3DXVECTOR3(XPos, YPos, ZPos),
                  &D3DXVECTOR3(0.0f, -1.0f, 0.0f),
                  &Hit, &FaceIndex, &u, &v, &Dist, NULL, NULL);

    // Если пересечение произошло, возвращаем ближайшую высоту снизу
    if(Hit == TRUE)
        return YPos - Dist;

    return YPos; // Если пересечения нет, возвращаем полученную высоту
}

float GetHeightAbove(ID3DXMesh *Mesh,
                    float XPos, float YPos, float ZPos)
{
    BOOL Hit;           // Флаг касания полигона
    float u, v, Dist;    // Рабочие переменные и расстояние
                        // до точки пересечения
    DWORD FaceIndex;    // Грань, которую пересекает луч

    // Проводим тест пересечения для сетки
    D3DXIntersect(Mesh,
                  &D3DXVECTOR3(XPos, YPos, ZPos),
                  &D3DXVECTOR3(0.0f, 1.0f, 0.0f),
                  &Hit, &FaceIndex, &u, &v, &Dist, NULL, NULL);

    // Если пересечение произошло, возвращаем ближайшую высоту сверху
    if(Hit == TRUE)
        return YPos + Dist;

    return YPos; // Если пересечения нет, возвращаем полученную высоту
}

```

У каждой из трех представленных выше функций есть свое особое предназначение. **GetClosestHeight** возвращает высоту полигона

(координату  $Y$ ), который находится ближе всего к указанной точке. Например, если вы проверяете точку в трехмерном пространстве (скажем, с координатой  $Y = 55$ ) и есть полигон на 10 единиц выше точки и другой полигон на 5 единиц ниже ее, то функция **GetClosestHeight** вернет 50 (потому что нижний полигон ближе к проверяемой точке).

**GetHeightAbove** и **GetHeightBelow** сканируют соответствующее направление (верх или низ) и возвращают высоту ближайшего полигона. Используя **GetHeightBelow**, можно определить где разместить ваши объекты (в плане высоты) в мире. При перемещении объекта вы изменяете его высоту, основываясь на высоте ландшафта под ним. Кроме того, вы можете определить не слишком ли круто наклонен полигон, чтобы перемещаться по нему. Чтобы увидеть этот метод в действии, взгляните на пример использования квадродеревьев и октодеревьев, находящийся на прилагаемом к книге CD-ROM.

### ***Быстрая проверка пересечения***

Идущие во все стороны лучи и многочисленные полигоны затрудняют поддержку быстрого темпа игры. Работа трехмерного движка замедляется, когда вы начинаете проверять столкновения между несколькими объектами. Поэтому вам необходим способ ускорить проверку столкновений.

Один из самых впечатляющих способов ускорения проверки столкновений из найденных мной (особенно, когда вы имеете дело с сетками, представленными в виде квадродеревьев и октодеревьев) заключается в поддержке нескольких сеток. Верно, разделив уровень на несколько сеток вы можете выполнять обнаружение столкновений только для тех сеток, которым это необходимо.

Например, попробуйте разделить уровень на три сетки: землю (для отслеживания высоты), стены и препятствия (для проверки столкновений, чтобы персонажи не ходили сквозь стены) и декорации (все дополнительные полигоны, которые служат только для украшения уровня). В требуемое время вы проводите проверку пересечения с соответствующей сеткой.

### ***Обнаружение столкновений в классе `cNodeTreeMesh`***

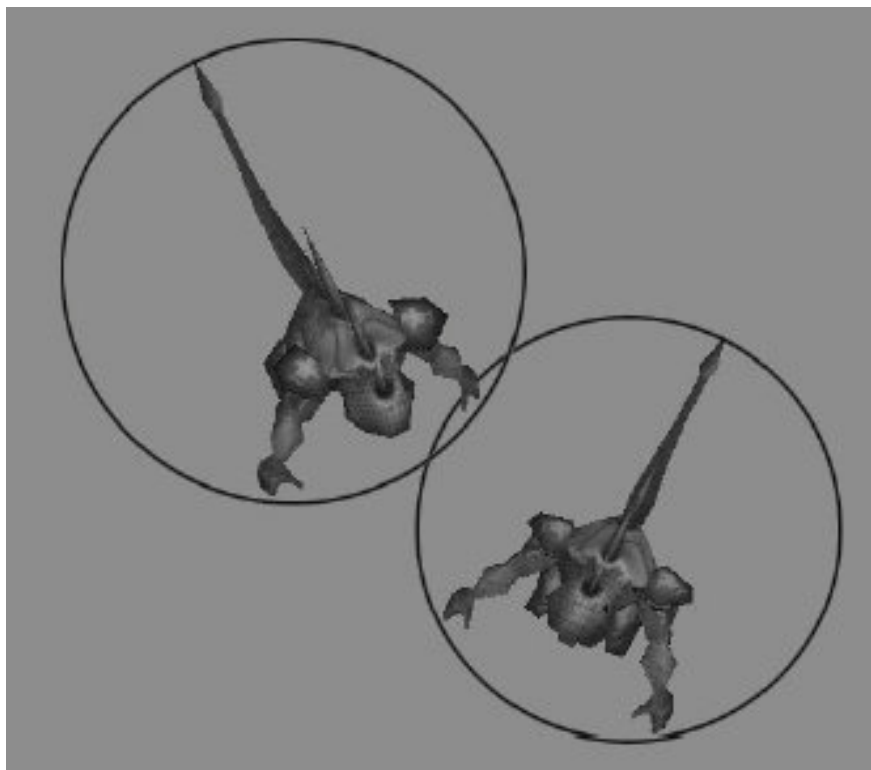
Чтобы усовершенствовать класс **cNodeTreeMesh**, можно добавить ранее упомянутые функции проверки пересечений и высоты. Теперь, используя пару простых функций, упакованных в замечательный класс, вы можете загрузить любую сетку уровня, и позволить персонажам ходить вокруг, ударяясь о стены и стоя на твердой земле (а не проваливаясь сквозь землю!).

## **Когда сталкиваются сетки**

Помимо определения того, когда сетки объектов сталкиваются с сеткой, образующей мир, вам надо знать, когда сталкиваются меньшие сетки. Вы же не хотите, чтобы ваши персонажи проходили друг друга насквозь, так что необходимо добавить обнаружение столкновений между объектами.

Вместо того, чтобы использовать проверку пересечения луча с плоскостью, как мы делали это в предыдущем разделе, обнаруживая столкновение с мировой сеткой, мы сократим обнаружение столкновений между объектами до одного простого вычисления. Помните ограничивающие сферы, обсуждавшиеся в разделе «Вычисление ограничивающей сферы»? Все, что вам надо сделать, это определить пересекаются ли ограничивающие сферы интересующих объектов.

Перед тем, как начать делать это, вам надо разобраться с несколькими вещами, включая недостатки использования ограничивающих сфер. Взгляните на рис. 8.9, где показаны два монстра. У них есть хвосты — очень длинные хвосты. Они влияют на общий размер ограничивающей сферы сетки — сферы будут большими, чтобы охватить всю сетку (включая хвост). Если вы будете передвигать двух монстров на рис 8.9, то заметите, что две ограничивающие сферы могут пересекаться, хотя сами монстры не касаются друг друга.



*Рис. 8.9. Сетки, содержащие выступающие части (такие, как хвосты монстров) могут при обнаружении столкновений использовать чрезмерно большие ограничивающие сферы*

Есть несколько способов решения проблемы чрезмерно большого радиуса ограничивающей сферы, и один из них вы можете реализовать достаточно быстро. Вместо использования ограничивающей сферы сетки, продвиньтесь чуть дальше и вычисляйте собственный радиус ограничивающей сферы для каждой сетки. Благодаря этому вы сможете быстро задать объем пространства, необходимый для безопасного покрытия сетки.

Оставив эту проблему в стороне, можно создать единую функцию, проверяющую пересекаются ли две ограничивающие сферы:

```
BOOL CheckSphereIntersect(
    float XCenter1, float YCenter1, float ZCenter1,
    float Radius1,
    float XCenter2, float YCenter2, float ZCenter2,
    float Radius2)
{
    float XDiff, YDiff, ZDiff, Distance;

    // Вычисляем расстояние между центрами
    XDiff = (float)fabs(XCenter2-XCenter1);
    YDiff = (float)fabs(YCenter2-YCenter1);
    ZDiff = (float)fabs(ZCenter2-ZCenter1);

    Distance = (float)sqrt(XDiff*XDiff+YDiff*YDiff+ZDiff*ZDiff);

    // Возвращаем TRUE, если две сферы пересекаются
    if(Distance <= (Radius1 + Radius2))
        return TRUE;

    // Нет пересечения
    return FALSE;
}
```

Если вызвать показанную функцию, передав ей координаты центра и радиусы двух ограничивающих сфер, она вернет **TRUE**, если сферы пересекаются, и **FALSE**, если сферы не пересекаются. Чтобы определить, пересекаются ли две сферы, мы вычисляем расстояние между их центрами, которое в случае пересечения должно быть меньше или равно сумме радиусов.

---

**СОВЕТ**

Чтобы оптимизировать функцию `CheckSphereIntersect`, можно убрать из кода вызов функции `sqrt`, как показано ниже:

```
Distance = XDiff*XDiff+YDiff*YDiff+ZDiff*ZDiff;
float RadiusDistance =
    (Radius1+Radius2)*(Radius1+Radius2)*3.0f;

// Возвращаем TRUE, если сферы пересекаются
if(Distance <= RadiusDistance)
    return TRUE;

return FALSE; // Нет пересечения
```

Расстояние сравнивается с произведением суммы радиусов без использования функции `sqrt` для вычисления действительного расстояния между центральными точками.

---

## Щелчки мыши и сетки

В последнем исследовании пересечений сеток мы сосредоточимся на возможности, которая наверняка потребуется вам при работе с трехмерной графикой: возможности щелкнуть по сетке и точно узнать, какая грань была выбрана. Вы уже знаете, как просканировать сетку, чтобы увидеть, какой полигон пересекает луч (посмотрите раздел «Испускание лучей» ранее в этой главе) — теперь вам надо сформировать луч, проходящий через позицию

курсора мыши в трехмерной сцене и посмотреть, на какой полигон указывает мышь.

Чтобы точно определить, по какому полигону сетки щелкнул игрок, помимо использования функции **D3DXIntersect** (которая была описана в разделе «Испускание лучей» ранее в этой главе), вы воспользуетесь ее аргументом **DWORD \*PFaceIndex**. Это указатель на переменную типа **DWORD**, которая будет содержать индекс грани, пересекаемой выпущенным вами лучом.

Обратите внимание, что если вы испускаете луч из воображаемой точки размещения зрителя (центра экрана) к курсору мыши, вам надо проверить на столкновение каждый полигон сцены. Ближайшая к зрителю точка пересечения (та, расстояние до которой меньше всего) и даст полигон, по которому щелкнул пользователь. Все это можно вычислить с помощью следующего кода (где используется код из примеров DirectX SDK):

```
// Graphics = ранее инициализированный объект cGraphics
// Mouse = ранее инициализированный объект мыши cInputDevice
D3DXVECTOR3 vecRay, vecDir; // Местоположение и направление луча
D3DXVECTOR3 v; // Временный вектор
D3DXMATRIX matProj, matView; // Матрицы проекции и вида
D3DXMATRIX m; // Временная матрица

// Получаем текущие преобразования проекции и вида
Graphics.GetDeviceCOM()->GetTransform(D3DTS_PROJECTION,
                                         &matProj);
Graphics.GetDeviceCOM()->GetTransform(D3DTS_VIEW,
                                         &matView);

// Инвертируем матрицу вида
D3DXMatrixInverse(&m, NULL, &matView);

// Фиксируем координаты мыши (подготавливаемся к чтению)
Mouse.Read();

// Вычисляем вектор луча выбора в экранном пространстве
v.x = (((2.0f * Mouse.GetXPos()) / Graphics.GetWidth()) -
        1) / matProj._11;
v.y = -(((2.0f * Mouse.GetYPos()) / Graphics.GetHeight()) -
        1) / matProj._22;
v.z = 1.0f;

// Преобразуем луч в экранном пространстве
vecRay.x = m._41;
vecRay.y = m._42;
vecRay.z = m._43;
vecDir.x = v.x*m._11 + v.y*m._21 + v.z*m._31;
vecDir.y = v.x*m._12 + v.y*m._22 + v.z*m._32;
vecDir.z = v.x*m._13 + v.y*m._23 + v.z*m._33;
```

Чтобы эффективно использовать представленный выше код, пойдете дальше и загрузим сетку в объект **ID3DXMesh**. Чтобы загрузка и хранение сетки были проще, можно использовать объект **cMesh** графического ядра:

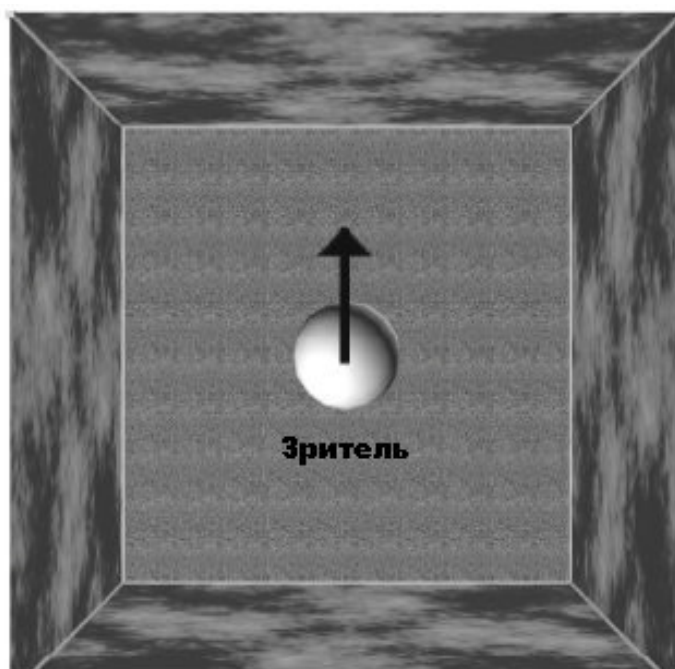
```
// Mesh = ранее загруженный объект cMesh
ID3DXMesh *pMesh;
```

```
BOOL      Hit;  
DWORD     Face;  
float     u, v, Dist;  
  
// Получаем указатель на объект ID3DXMesh из cMesh  
pMesh = Mesh->GetParentMesh()->m_Mesh;  
  
// Вызываем показанный ранее код для получения  
// векторов луча и проверяем пересечение  
  
ID3DXIntersect(pMesh, &vecRay, &vecDir, &Hit, &Face,  
               &u, &v, &Dist, NULL, NULL);  
  
// Если Hit равно TRUE, пользователь щелкнул по сетке
```

Определяя пересечение с наименьшим расстоянием до зрителя (используя показанный код), вы проверяете каждую сетку сцены (или другие сетки, например, персонажей) и находите по какой из них был выполнен щелчок.

## Использование небесного куба

*Небесный куб (sky box)* — это графическая техника, в которой текстурированный трехмерный куб окружает зрителя. При визуализации небесного куба его центр всегда располагается в точке местоположения зрителя, чтобы пользователь видел внутренние текстурированные грани куба. Данная техника позволяет имитировать мир, окружающий пользователя (рис. 8.10).



**Рис. 8.10.** Небесный куб создает у зрителя иллюзию, что его окружает огромный мир

Реализовать небесный куб просто. Вам нужна только сетка куба (у которой грани обращены внутрь). Для хранения сетки куба замечательно подойдет буфер вершин. Что касается текстур, то их вам потребуется

шесть — по одной для каждой стороны. Сетка не должна быть очень большой, достаточно куба с размером стороны 20.0 единиц. Размер текстур должен быть  $256 \times 256$  или больше, поскольку меньшие будут выглядеть растянутыми и некрасивыми.

## Создание класса небесного куба

Лучший способ реализации небесного куба в вашей игре — создание класса, который легко можно включать в собственные проекты. Этот класс, в нашем случае названный **cSkyBox**, управляет каждым аспектом небесного куба — от создания буфера вершин, содержащего данные куба, до хранения используемых при визуализации текстур. Приведенный ниже код класса небесного куба очень компактен и будет хорошим дополнением к любой игровой библиотеке:

```
class cSkyBox
{
private:
    typedef struct sSkyBoxVertex {
        float x, y, z;
        float u, v;
    } sSkyBoxVertex;
    #define SkyBoxFVF (D3DFVF_XYZ | D3DFVF_TEX1)

    cGraphics *m_Graphics; // Родительский объект cGraphics
    cTexture m_Textures[6]; // Текстуры граней (0-5)
    cVertexBuffer m_VB; // Буфер вершин сетки
    cWorldPosition m_Pos; // Ориентация небесного куба

public:
    cSkyBox(); // Конструктор
    ~cSkyBox(); // Деструктор

    // Создание и освобождение объекта класса небесного куба
    BOOL Create(cGraphics *Graphics);
    BOOL Free();

    // Установка текстуры для заданной грани. Позволяет
    // использовать прозрачность и менять формат хранения
    BOOL LoadTexture(short Side, char *Filename,
        D3DCOLOR Transparent = 0,
        D3DFORMAT = D3DFMT_UNKNOWN);

    // Абсолютное и относительное вращение куба
    BOOL Rotate(float XRot, float YRot, float ZRot);
    BOOL RotateRel(float XRot, float YRot, float ZRot);

    // Визуализация небесного куба (с необязательным
    // альфа-смешиванием) с использованием текущего
    // преобразования вида из Camera.
    BOOL Render(cCamera *Camera, BOOL Alpha = FALSE);
};
```

В последующих разделах мы подробно рассмотрим каждую функцию из приведенного кода.



**cSkyBox::Create и cSkyBox::Free**

Этот дуэт функций вы используете для получения родительского объекта графической системы, создания сетки небесного куба и освобождения всех используемых объектов **cTexture**. Взгляните на код:

```

BOOL cSkyBox::Create(cGraphics *Graphics)
{
    sSkyBoxVertex Verts[24] = {
        { -10.0f, 10.0f, -10.0f, 0.0f, 0.0f }, // Верхние вершины
        { 10.0f, 10.0f, -10.0f, 1.0f, 0.0f },
        { -10.0f, 10.0f, 10.0f, 0.0f, 1.0f },
        { 10.0f, 10.0f, 10.0f, 1.0f, 1.0f },
        { -10.0f, -10.0f, 10.0f, 0.0f, 0.0f }, // Нижние вершины
        { 10.0f, -10.0f, 10.0f, 1.0f, 0.0f },
        { -10.0f, -10.0f, -10.0f, 0.0f, 1.0f },
        { 10.0f, -10.0f, -10.0f, 1.0f, 1.0f },
        { -10.0f, 10.0f, -10.0f, 0.0f, 0.0f }, // Левые вершины
        { -10.0f, 10.0f, 10.0f, 1.0f, 0.0f },
        { -10.0f, -10.0f, -10.0f, 0.0f, 1.0f },
        { -10.0f, -10.0f, 10.0f, 1.0f, 1.0f },
        { 10.0f, 10.0f, 10.0f, 0.0f, 0.0f }, // Правые вершины
        { 10.0f, 10.0f, -10.0f, 1.0f, 0.0f },
        { 10.0f, -10.0f, 10.0f, 0.0f, 1.0f },
        { 10.0f, -10.0f, -10.0f, 1.0f, 1.0f },
        { -10.0f, 10.0f, 10.0f, 0.0f, 0.0f }, // Передние вершины
        { 10.0f, 10.0f, 10.0f, 1.0f, 0.0f },
        { -10.0f, -10.0f, 10.0f, 0.0f, 1.0f },
        { 10.0f, -10.0f, 10.0f, 1.0f, 1.0f },
        { 10.0f, 10.0f, -10.0f, 0.0f, 0.0f }, // Задние вершины
        { -10.0f, 10.0f, -10.0f, 1.0f, 0.0f },
        { 10.0f, -10.0f, -10.0f, 0.0f, 1.0f },
        { -10.0f, -10.0f, -10.0f, 1.0f, 1.0f },
    };

    Free(); // Освобождаем предыдущий небесный куб

    // Проверка ошибок
    if((m_Graphics = Graphics) == NULL)
        return FALSE;

    // Создаем буфер вершин
    // (и копируем в него вершины небесного куба)
    if(m_VB.Create(m_Graphics, 24, SkyBoxFVF,
        sizeof(sSkyBoxVertex)) == TRUE)
        m_VB.Set(0, 24, (void*)&Verts);

    // Поворачиваем небесный куб в положение по умолчанию
    Rotate(0.0f, 0.0f, 0.0f);

    return TRUE; // Возвращаем флаг успеха!
}

BOOL cSkyBox::Free()
{
    m_Graphics = NULL; // Очищаем родительский объект cGraphics

    for(short i = 0; i < 6; i++) // Освобождаем текстуры
        m_Textures[i].Free();
}

```

```

    m_VB.Free(); // Освобождаем буфер вершин

    return TRUE; // Возвращаем флаг успеха!
}

```

Как видите, названия функций соответствуют их сути. Сначала **Create** создает буфер вершин (для сетки куба с 12 гранями и 6 сторонами). Этот буфер вершин заполняется информацией, заданной внутри функции **Create**. После создания буфера вершин функция **Create** продолжает работу, и задает ориентацию сетки небесного куба, устанавливая ее в положение по умолчанию.

Используя вращение, вы можете создавать слои из небесных кубов для реализации трехмерных эффектов. Например, представьте себе слоеный пирог из нескольких небесных кубов — один со звездами, другой с облаками, третий с солнцем и луной — и так вы создаете законченное вращающееся небо со следующими по своим орбитам солнцем и луной!

Закончив с функцией **Create**, я перехожу к функции **Free**. Она освобождает ресурсы текстур и буфер вершин, используя функции **Free** соответствующих объектов.

### **cSkyBox::LoadTexture**

Показанная ниже функция **LoadTexture** загружает единственную текстуру, которая будет использоваться для текстурирования одной стороны небесного куба.

```

BOOL cSkyBox::LoadTexture(short Side, char *Filename,
                          D3DCOLOR Transparent, D3DFORMAT Format)
{
    // Проверка ошибок
    if(m_Graphics == NULL || Side < 0 || Side > 5)
        return FALSE;

    m_Textures[Side].Free(); // Освобождаем предыдущую текстуру

    return m_Textures[Side].Load(m_Graphics, Filename,
                                Transparent, Format);
}

```

Функция **LoadTexture** загружает одно растровое изображение в указанную текстуру (используя заданные прозрачный цвет и формат хранения, если вы намереваетесь применять копирование с учетом прозрачности или альфа-смешивание). Следует обратить внимание на нумерацию граней небесного куба. Соответствие номеров и граней приведено в таблице 8.2.

Таблица 8.2. Номера граней для `cSkyBox::LoadTexture`

Значение	Грань
0	Верхняя
1	Нижняя
2	Левая
3	Правая
4	Передняя
5	Задняя

### ***cSkyBox::Rotate* и *cSkyBox::RotateRel***

Вы уже узнали, что небесные кубы могут вращаться, создавая иллюзию, что небесные объекты движутся по орбитам вокруг зрителя. Рассматриваемые две функции позволяют менять поворот небесных кубов:

```

BOOL cSkyBox::Rotate(float XRot, float YRot, float ZRot)
{
    return m_Pos.Rotate(XRot, YRot, ZRot);
}

BOOL cSkyBox::RotateRel(float XRot, float YRot, float ZRot)
{
    return m_Pos.RotateRel(XRot, YRot, ZRot);
}

```

Чтобы изменить значения вращения эти две функции вызывают методы класса **cWorldPosition** графического ядра (дополнительную информацию об объекте **cWorldPosition** вы найдете в главе 6, «Создаем ядро игры»).

### ***cSkyBox::Render***

Вот и действительно важная функция. Она центрирует небесный куб относительно указанной камеры, включает альфа-смешивание и альфа-проверку (если необходимо) и визуализирует шесть сторон небесного куба (или как минимум те из них, которые текстурированы):

```

BOOL cSkyBox::Render(cCamera *Camera, BOOL Alpha)
{
    D3DXMATRIX matWorld;
    short      i;

    // Проверка ошибок
    if(m_Graphics == NULL || Camera == NULL)
        return FALSE;

    // Располагаем небесный куб вокруг зрителя
    m_Pos.Move(Camera->GetXPos(),
               Camera->GetYPos(),
               Camera->GetZPos());
    m_Graphics->SetWorldPosition(&m_Pos);
}

```

```

// Включаем альфа-проверку и альфа-смешивание
m_Graphics->EnableAlphaTesting(TRUE);
if(Alpha == TRUE)
    m_Graphics->EnableAlphaBlending(TRUE, D3DBLEND_SRCCOLOR,
                                     D3DBLEND_DESTCOLOR);

// Рисуем каждый слой
for(i = 0; i < 6; i++) {
    if(m_Textures[i].IsLoaded() == TRUE) {
        m_Graphics->SetTexture(0, &m_Textures[i]);
        m_VB.Render(i*4, 2, D3DPT_TRIANGLESTRIP);
    }
}

// Отключаем альфа-проверку и альфа-смешивание
m_Graphics->EnableAlphaTesting(FALSE);
if(Alpha == TRUE)
    m_Graphics->EnableAlphaBlending(FALSE);

return TRUE;
}

```

При вызове функции **Render** вы должны передать ей текущий объект **cCamera**, который используется для визуализации сцены. Необязательному аргументу **Alpha** вы можете присвоить значение **TRUE**, чтобы функция **Render** визуализировала небесный куб с использованием техники альфа-смешивания.

## Использование класса небесного куба

Теперь пора посмотреть небесный куб в действии. Вообще-то, если вы уже смотрели пример **NodeTree**, то видели небесный куб. В этом примере для небесного куба используется единственная текстура (звезды). Хотя я и использовал такой простой пример, в небесном кубе можно применять до шести текстур, что позволяет создавать изумительно выглядящие трехмерные сцены.

## Заканчиваем с трехмерной графикой

Вам редко будет нужен более мощный движок, чем представленный в этой главе. Благодаря двум способам визуализации ваших уровней и методу быстрого определения видимых объектов, частота кадров в вашей игре будет достаточно высокой.

### Программы на CD-ROM

Программы, демонстрирующие обсуждавшийся в этой главе код, находятся на прилагаемом к книге CD-ROM. В папке \BookCode\Chap08\ вы найдете следующие программы:

**MeshLvl** — показывает, как использовать отдельные сетки для конструирования уровней. Местоположение: \BookCode\Chap08\MeshLvl\.

**NodeTree** — демонстрирует визуализацию уровней, представленных в виде древовидной структуры. Местоположение: \BookCode\Chap08\NodeTree\.

**Objects** — показывает, как использовать пирамиду видимого пространства для отбрасывания находящихся вне поля зрения объектов. Местоположение: \BookCode\Chap08\Objects\.

# Глава 9

## Объединение двухмерной и трехмерной графики

Когда дело доходит до графики, нет ни ограничений, ни правил, которых вы должны обязательно придерживаться; так что вы не обязаны использовать только двухмерную или только трехмерную графику. Можете безнаказанно смешивать их вместе. Эта глава будет вашим руководством в понимании трюков, лежащих в основе смешивания двух измерений для достижения сногшибательных эффектов.

В этой главе вы узнаете следующее:

- Использование двухмерной графики в трехмерном движке.
- Рисование трехмерных объектов в двухмерном мире.

### Смешивание двух измерений

Представьте себе трехмерных персонажей, перемещающихся по двухмерным изображениям или использование двухмерных блоков в трехмерной сцене. Я знаю, что вам смертельно хочется узнать как использовать трехмерных персонажей с плоскими двухмерными изображениями, но вы можете недоумевать, зачем использовать двухмерные блоки в трехмерном мире.

Подумайте о следующем: использование щитов (двухмерных объектов) экономит полигоны, необходимые для представления простых объектов в вашем трехмерном мире — таких, как деревья, камни и т.д. Все эти объекты добавляют красоты вашей игре.

Фактически, если вы наблюдательны, то возможно видели, что многие трехмерные игры используют двухмерные объекты, причем таким способом, что эффект легко заметить. Например, в игре Mario 64 для N64 (игровой приставки) вы можете подойти к дереву и обойти вокруг него. Обратите внимание, что дерево не вращается. Это потому что оно двухмерное.

Независимо от причин для смешивания двухмерной и трехмерной графики, вам будет полезно взглянуть на пару программ, которые я создал, чтобы помочь вам на этом пути. Первая из них, названная 2Din3D рисует

трехмерную сетку, представляющую уровень игры, и двумерные блоки поверх нее, представляющие игровые объекты.

## Использование двумерных объектов в трехмерном мире

В главе 2, «Рисование с DirectX Graphics», обсуждалось использование щитов (двухмерных объектов, выравниваемых таким образом, что они все время направлены на зрителя), а в этой главе я расскажу об использовании двумерных объектов, таких как блоки, в трехмерных сценах.

Если хотите увидеть двумерные объекты в реальной игре, посмотрите на Paper Mario для N64. Марио (и все другие персонажи игры) рисуются как плоские изображения с использованием техники щитов (текстуры, наложенные на плоские полигоны). Весь окружающий Марио мир трехмерен, и при перемещении по нему трехмерная сцена сдвигается, чтобы всегда показывать всех игровых персонажей под правильным углом.

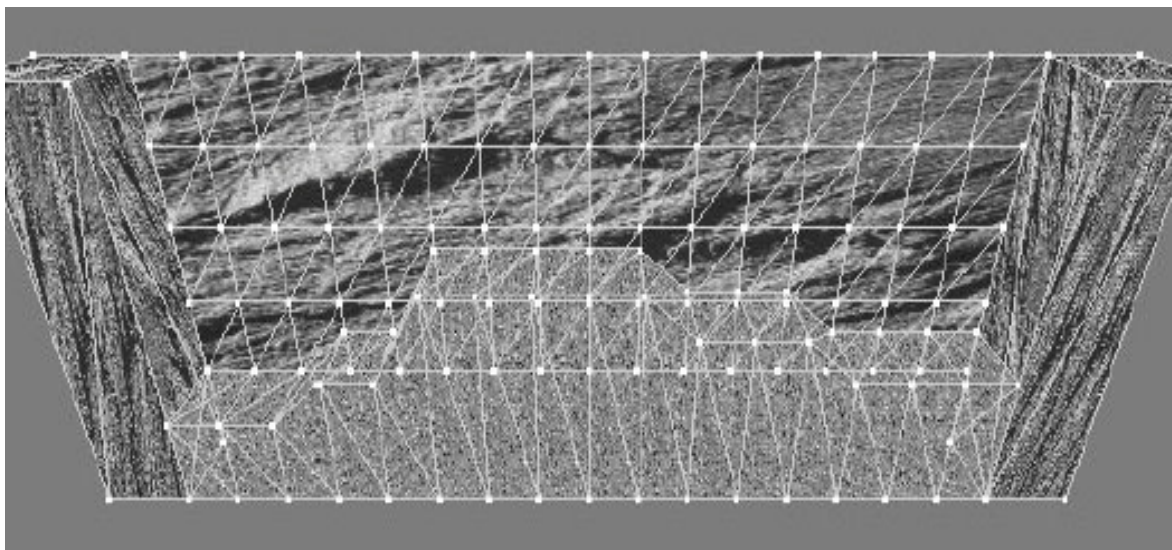
Вы можете воспроизвести такой эффект, используя щиты для имитации двумерных объектов, либо используя двумерную блочную графику. Со щитами достаточно легко добавить глубину (когда объекты, находящиеся далеко от зрителя выглядят меньше). При использовании двумерных блоков такого же эффекта можно достичь с помощью масштабирования, но зачем беспокоиться, когда у нас есть щиты?

### Рисование блоков в трехмерном мире

Хотя объединение двумерных блоков с трехмерными сетками может казаться сложным, ничего нового в нем нет. О том, как загружать и рисовать двумерные блоки вы узнали в главе 7, «Использование двумерной графики», а о том, как работать с трехмерными сетками — в главе 2. В этой главе вы узнаете как объединить двумерные и трехмерные компоненты, чтобы рисовать трехмерную сетку, представляющую уровень игры, и наложить на сетку двумерные графические блоки, представляющие игровые объекты.

Сейчас вам потребуется трехмерная сетка, представляющая уровень игры. Такая сетка показана на рис. 9.1 и находится на прилагаемом к книге CD-ROM (в папке \BookCode\Chap09\2Din3D). Там же вы найдете растровое изображение, содержащее набор двумерных блоков, используемых для рисования игрового персонажа.

Обычно персонаж перемещается по трехмерному миру (зачастую вдоль единственной оси координат), и при перемещении персонажа передвигается и камера. Камера должна располагаться на небольшом расстоянии от персонажа и чуть выше его, чтобы полностью проявлялся эффект трехмерности уровня. Чтобы увидеть, о чем я говорю, взгляните на рис. 9.2, где показано приложение 2Din3D в действии.



*Рис. 9.1. Большая сетка представляет трехмерный уровень в котором вы будете рисовать двухмерные объекты*



*Рис. 9.2. Программа 2Din3D демонстрирует использование двухмерных блоков, в смеси со сложным трехмерным уровнем. На рисунке вы видите двухмерный блок (монстра), нарисованный поверх трехмерной сетки, представляющей уровень*

Позвольте без дальнейших разглагольствований обратить ваше внимание на код, который загружает представляющую уровень сетку, создает набор двухмерных блоков, представляющих объект (например, персонаж игрока) и рисует все это в каждом кадре с правильным выравниванием. Начнем с загрузки сетки, представляющей уровень.



### **Загрузка сетки уровня**

Графическое ядро является библиотекой полезных функций, и мы воспользуемся им для загрузки X-файла, содержащего сетку уровня. Объект графического ядра, который выполняет загрузку X-файлов, это **cMesh**. После того, как сетка загружена в объект **cMesh**, вы создаете объект **cObject**, используемый для визуализации сетки уровня. В данном примере используется сетка **Level.x**, которую вы найдете на прилагаемом к книге CD-ROM (в папке `\BookCode\Chap09\2Din3D`).

```
// Graphics = ранее инициализированный объект cGraphics
cMesh LevelMesh; // Сетка уровня
cObject LevelObj; // Объект, используемый для визуализации

// Загружаем X-файл с именем Level.x
LevelMesh.Load(&Graphics, "Level.x");

// Назначаем сетку объекту
LevelObj.Create(&Graphics, &LevelMesh);
```

Использовать графическое ядро действительно удобно; показанный выше код — это все, что нам надо. Далее следует загрузка блоков.

### **Загрузка блоков**

Двухмерные блоки в данном примере содержат серию изображений, используемых для рисования анимированного идущего монстра. Эти блоки объединены в одно растровое изображение, загружаемое в объект **cTexture**. Изображение находится в файле **Tiles.bmp** на прилагаемом к книге CD-ROM (также в папке `\BookCode\Chap09\2Din3D`).

```
// Graphics = Ранее инициализированный объект cGraphics
cTexture Tiles;

// Загружаем текстуру, указав, что цвет 0 - прозрачный
Tiles.Load(&Graphics, "Tiles.bmp", 0, D3DFMT_A1R5G5B5);
```

Вы почти готовы начать рисование уровня — сейчас загружены сетка и набор блоков. Однако, перед рисованием сцены, надо выполнить несколько подготовительных действий.

### **Подготовка к рисованию**

Перед визуализацией каждого кадра вы должны выровнять камеру относительно игрока и очистить экранный буфер (и Z-буфер).

```
// Graphics = ранее инициализированный объект cGraphics
// XPos, YPos, ZPos = координаты персонажа по которому
//                      выравнивается камера
cCamera Camera;

// Очищаем экран и Z-буфер
Graphics.Clear();
```

```
// Выравниваем камеру
// (располагая ее перед персонажем и чуть выше)
Camera.Point(XPos, YPos + 50.0f, ZPos - 500.0f,
             XPos, YPos, ZPos);
Graphics.SetCamera(&Camera);
```

Вы почти закончили; теперь можно рисовать сетку уровня.

## Рисование сетки уровня

Вызов **cObject::Render** позаботится о визуализации сетки уровня. Как и все вызовы визуализации, вы должны поместить его между вызовами **cGraphics::BeginScene** и **cGraphics::EndScene**.

```
if(Graphics::BeginScene()==TRUE) {
    LevelObj.Render();
    Graphics::EndScene();
}
```

Теперь уровень визуализирован и осталось только нарисовать двухмерные блоки.

## Рисование двухмерных объектов

Вы уже готовы нарисовать на экране блоки в соответствующих им позициях. Для этого вы в цикле перебираете все объекты и рисуете их на экране. Профессиональные программисты игр могут даже использовать пирамиду видимого пространства, чтобы отбрасывать объекты, находящиеся вне поля зрения. Представленный пример основывается на коде из предыдущих глав и рисует один объект — персонаж игрока.

### ПРИМЕЧАНИЕ

В использовании двухмерных блоков есть один недостаток. Вы не сможете использовать Z-буфер (по крайней мере, если так же как и я используете интерфейс **ID3DXSprite**), а это значит, что вы не можете просто нарисовать блок и ожидать, что правильное смешивание с трехмерной сценой произойдет автоматически. Нарисованный двухмерный блок будет полностью видим, пока вы не нарисуете поверх него что-нибудь еще (полигон или другой блок).

```
// Graphics = ранее инициализированный объект cGraphics
// Начинаем сцену
if(Graphics::BeginScene()==TRUE) {

    // Рисуем сетку уровня
    LevelObj.Render();
```

Теперь уровень визуализирован и можно рисовать двухмерные объекты. Включим альфа-проверку, чтобы прозрачные области блоков отсекались и выключим Z-буферизацию, чтобы ничто не скрывало блоки.

```
// Разрешаем прозрачность
Graphics.EnableAlphaTesting(TRUE);
```

```
// Запрещаем Z-буферизацию
Graphics.EnableZBuffer(FALSE);

// Начинаем рисовать спрайты
Graphics.BeginSprite();

// Рисуем объект персонажа игрока ( блок размером 64 x 64)
// основываясь на координатах середины нижней стороны блока.
// Поскольку камера центрируется по объекту, вам достаточно
// нарисовать блок в центре экрана (сместив на половину размера
// блока). Обратите внимание, что 640 и 480 - это размеры экрана
Tiles.Blit(0, 0, (640-64)/2, 480/2-64, 64, 64);

// Заканчиваем рисование спрайтов
Graphics.EndSprite();

// Отключаем альфа-проверку
Graphics.EnableAlphaTesting(FALSE);

// Завершаем рисование сцены
Graphics::EndScene();
}
```

### Перемещение в трехмерном мире

У неподвижных картинок очень мало применений, так что необходимо добавить к программе немного движения; например, пусть персонаж сможет ходить по уровню. Поскольку персонаж представлен набором трехмерных координат, а уровень является объектом **ID3DXMesh**, почему бы не воспользоваться функцией **ID3DXIntersect** для проверки изменения высоты и столкновений персонажа со стенами? Фактически, почему бы не использовать функцию проверки столкновений? Ничто не мешает сделать это, так что двигайтесь вперед и вызывайте функции, разработанные в главе 8, «Создание трехмерного движка».

---

#### ПРИМЕЧАНИЕ

Чтобы увидеть, как я добавил проверку пересечений, взгляните на полный исходный код демонстрационной программы 2Din3D, находящейся на прилагаемом к книге CD-ROM (в папке \BookCode\Chap09\2Din3D\). Там вы также найдете некоторые рудиментарные возможности перемещения. В программе вы можете использовать клавиши управления курсором для перемещения по уровню. Наслаждайтесь!

---

### Добавление трехмерных объектов к двумерному миру

В таких играх, как Final Fantasy и Parasite Eve, созданных Square Co. Ltd, вы можете наслаждаться красотой предварительно визуализированных фоновых изображений и в то же время использовать трехмерные модели игровых объектов. Смешивание трехмерной и двумерной графики является наиболее охраняемым секретом игровых компаний, и эту тайну стоит раскрыть.

Если вы никогда не видели трехмерную графику в двухмерном движке, о которой я говорю, взгляните на рис. 9.3, где показан результат работы движка, находящегося на прилагаемом к книге CD-ROM (в папке \BookCode\Chap09\3Din2D). На иллюстрации изображен статический двухмерный фон, поверх которого расположен трехмерный объект. Трехмерный объект может свободно перемещаться по фону так, будто тот тоже трехмерный.



*Рис. 9.3. Движок, совмещающий трехмерную и двухмерную графику, позволяет использовать замечательно выглядящий двухмерный фон вместе с трехмерными элементами, такими как персонажи игры*

Фон — это предварительно визуализированное растровое изображение, которое отображается в каждом кадре с использованием техники двухмерного копирования, обсуждавшейся в главе 7. Если более точно, объект **ID3DXSprite** выполняет рисование растрового изображения (загруженного в объект **IDirect3DTexture9**). Для трехмерных объектов движка замечательно подходит объект **ID3DXMesh**, не говоря уж о классах графического ядра **cMesh** и **cObject**, которые помогут вам и при загрузке и при отображении.

Итак, с одной стороны у вас есть предварительно визуализированный фон, помещаемый на экран в каждом кадре, а с другой стороны у вас есть трехмерные объекты, рисуемые в сцене. Кажется все просто? Нет, по крайней мере на первый взгляд.

Как извлечь информацию о глубине из двухмерного изображения? Фактически, для этого есть несколько способов. Вот некоторые из них:

- Создаем предварительно визуализируемый фон в программе трехмерного моделирования, такой как gameSpace Light или 3D Studio Max и сохраняем изображение вместе с буфером глубины, содержащим Z-значения для каждого пикселя. В каждом кадре игры копируем буфер глубины изображения в буфер глубины вторичного буфера и потом рисуем трехмерные объекты.
- Создание фона из слоев. Вы начинаете с нижнего слоя и последовательно рисуете каждое изображение, а в соответствующем слое рисуете трехмерных персонажей; таким образом последовательные слои закрывают части нижележащих слоев (и трехмерных объектов).
- Использование высокодетализированных заранее визуализированных фоновых изображений и упрощенной версии той сетки, которая использовалась вами при визуализации сцены в программе трехмерного моделирования. Эта сетка используется для визуализации Z-значений и обнаружения столкновений. В этом случае трехмерные объекты могут полагаться на Z-буфер, для рисования на правильной глубине.

Здесь вам представлены три возможных способа получения информации о трехмерной глубине из двухмерного изображения. Первый вариант, хранение изображения в формате включающем Z-значение для каждого пикселя, выглядит заманчиво, и он был бы самым лучшим вариантом, вот только на данный момент его нельзя использовать с DirectX.

Хотя DirectX Graphics и предоставляет минимальную функциональность для блокировки буфера глубины и управления им (обратите внимание, что DirectX не позволяет копировать данные в буфер глубины), у вас нет никаких гарантий, что все видеокарты будут поддерживать механизм блокировки буфера глубины. Кроме того, ручная блокировка и разблокировка буфера глубины в каждом кадре значительно повышают требования к производительности системы.

Вариант номер два — рисование фона по слоям. Во многих случаях этот вариант самый легкий для использования. Разделив фон на части, вы можете рисовать их в заданном порядке, добавляя в каждом слое необходимые трехмерные объекты. Если в вашем фоне нет областей, которые могут скрывать трехмерные объекты, можно просто нарисовать все слои сразу, а затем нарисовать трехмерные объекты.

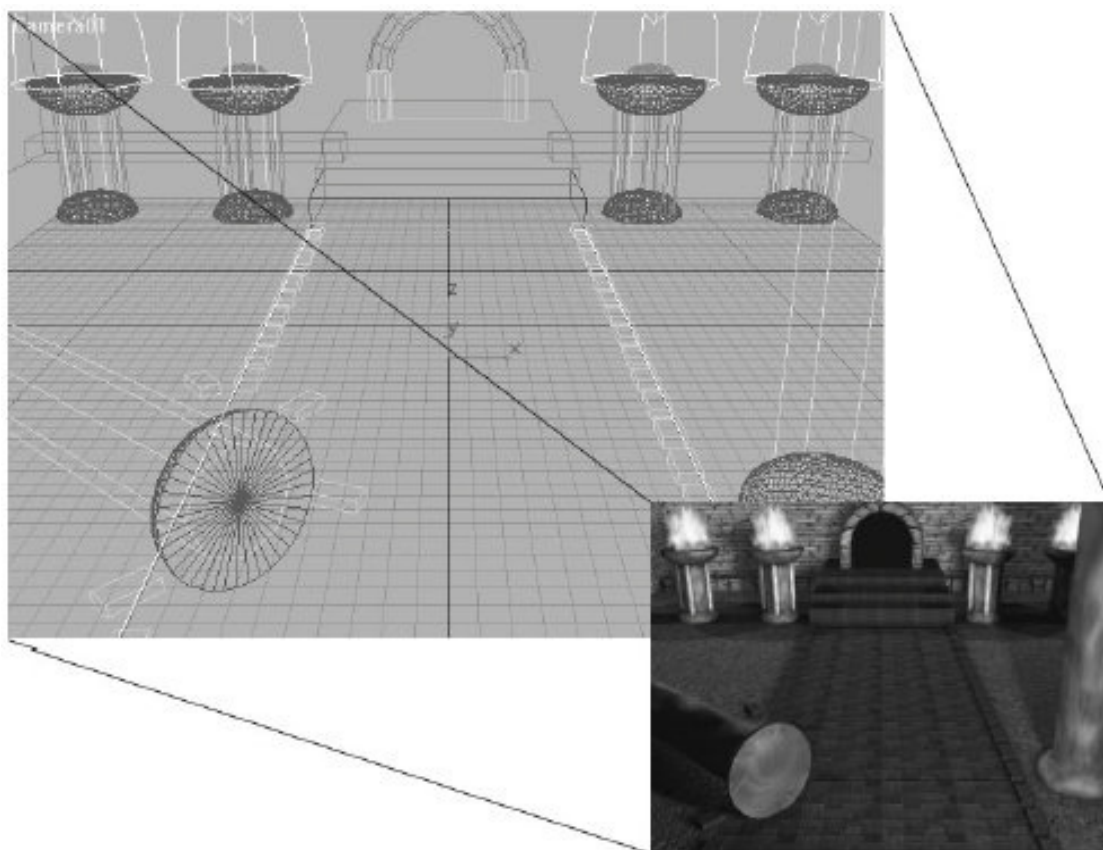
Последний, третий, вариант заключается в использовании упрощенной версии той сетки, которая применялась для визуализации фона, чтобы

создать Z-значения для каждого пикселя. Именно этот вариант я использую в книге. Поскольку сцена предварительно визуализируется в программе трехмерного моделирования, вы можете взять упрощенную версию сетки сцены и использовать ее для заполнения Z-буфера в каждом кадре, потом скопировать изображение фона и нарисовать трехмерные объекты. Кроме того, вы можете использовать сетку для обнаружения столкновений (и для определения высоты, как это делалось в главе 8 при использовании движка NodeTree).

Теперь вы готовы конструировать программу 3Din2D; а начнем мы с получения изображения фона и сетки, с которой будем работать.

## Работа с двухмерным фоном

Как я упоминал в предыдущем разделе, вы разрабатываете двухмерный фон с помощью программы трехмерного моделирования, такой как gameSpace Light (а не в графическом редакторе, поскольку вам понадобятся данные о полигонах из программы моделирования). На рис. 9.4 представлены простая сетка и итоговый результат визуализации.



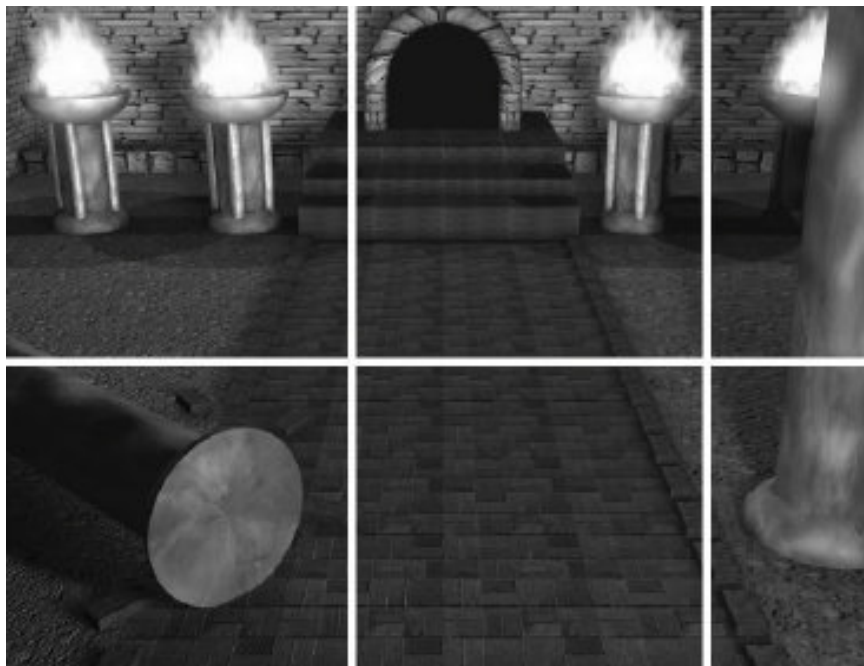
*Рис. 9.4. Вы создаете сетку в программе трехмерного моделирования и визуализируете фон*

Разрабатывая сцену вы можете сколько хотите увлекаться деталями, поскольку единственная вещь, которая реально необходима — это полученное в результате визуализации изображение (хотя полигоны, закрывающие вид, потребуют небольшой хитрости, как вы увидите в следующем разделе, «Работа с сеткой сцены»). Сейчас просто запомните



точное местоположение и ориентацию камеры, которые использовались при визуализации сцены (они пригодятся вам позже).

Когда смоделированная сцена визуализирована, вы сохраняете ее на диск в виде растрового изображения. Файл с растровым изображением необходимо разделить на текстуры меньшего размера. На рис. 9.5 снова показан пример фона, но на этот раз разделенного на шесть небольших прямоугольных кусков (текстур).



*Рис. 9.5. Фон разделен на шесть небольших прямоугольных кусков*

#### ПРИМЕЧАНИЕ

Если чувствуете себя храбрым, попробуйте создать класс, который загружает растровое изображение произвольного размера и делит его на фрагменты. Благодаря этому вам не потребуется обращаться к графическому редактору для разрезания изображения, что упростит загрузку изображений с различным разрешением.

Вам необходимо разделить фоновое изображение на рис. 9.5 на несколько текстур, чтобы Direct3D смог обработать их. В данном случае размеры фонового изображения  $640 \times 480$ , так что размеры текстур  $256 \times 256$  (для фрагментов 1, 2, 4 и 5) и  $128 \times 256$  (для фрагментов 3 и 6). Обратите внимание, что если фрагмент недостаточно большой, вам необходимо увеличить его, чтобы он соответствовал допустимому размеру текстуры (например, фрагменты 4, 5 и 6 должны быть увеличены до высоты в 256 пикселей).

Разделив изображение фона на шесть частей, сохраните каждый фрагмент в отдельном файле. Позднее вы загрузите шесть файлов в объекты **cTexture** графического ядра, которые будут использоваться для визуализации фрагментов на экране. Если предположить, что фрагменты нумеруются с 1 до 6, и в имени файла используется префикс Scene, для загрузки текстур можно использовать следующий код:

```
// Graphics = ранее инициализированный объект cGraphics
char    Filename[81]; // Имя файла загружаемой текстуры
cTexture Textures[6]; // Объекты для хранения текстур

for(short i = 0; i < 6; i++) {

    // Создаем имя файла с текстурой
    sprintf(Filename, "Scene%u.bmp", i+1);

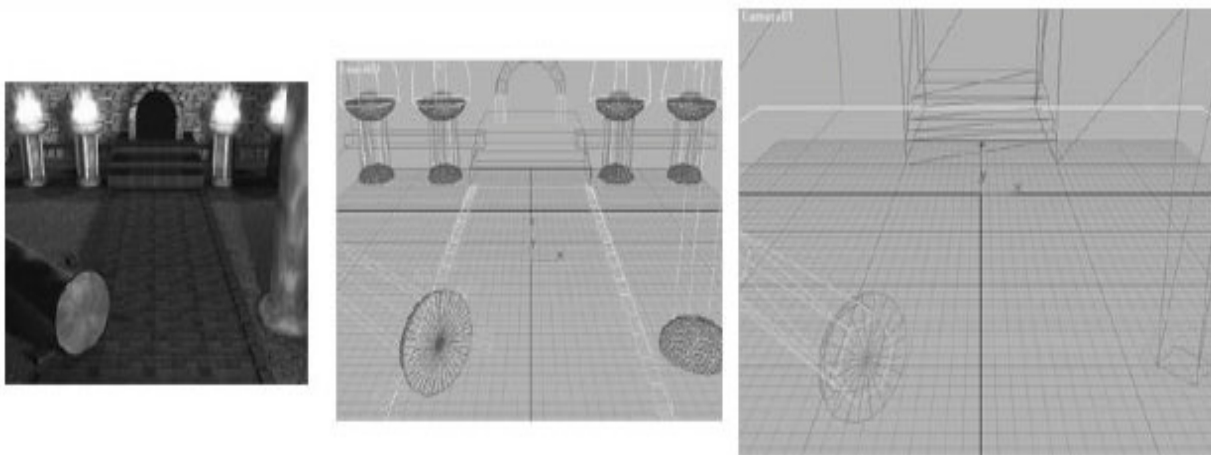
    // Загружаем текстуру
    Textures[i].Load(&Graphics, Filename);
}
```

Когда вы загрузили каждую из шести частей в соответствующий объект текстуры, можно использовать функцию **cTexture::Blit** для рисования их на экране. О том, как это сделать, будет рассказано в разделе «Визуализация сцены» дальше в этой главе, но перед этим сосредоточим внимание на сетке сцены.

## Работа с сеткой сцены

Ваш детализированный уровень выглядит замечательно, и теперь вы хотите добавить к нему какие-нибудь трехмерные объекты. Однако, сперва вам надо сконструировать упрощенную версию вашей сцены, которая будет использоваться в двух целях — для заполнения буфера глубины в каждом кадре, чтобы трехмерные объекты правильно смешивались с двухмерным фоном, и для обнаружения столкновений при перемещении объектов.

Когда я говорю «упрощенная», я действительно подразумеваю упрощение. Поскольку для создания Z-значений сетка должна будет визуализироваться в каждом кадре, то чем меньше полигонов вы используете, тем лучше. Однако, вам требуется достаточное количество полигонов, чтобы гарантировать корректное смешивание для трехмерных объектов. Чтобы увидеть, о чем я говорю, взгляните на рис. 9.6, где показано визуализированное изображение сцены, исходная сетка сцены и ее упрощенный вариант.



**Рис. 9.6.** Визуализированное изображение, используемое в качестве фона (слева). Исходная сетка, применявшаяся для визуализации фона в программе трехмерного моделирования (в центре). Упрощенная сетка, используемая для Z-буферизации и проверки пересечений (справа)

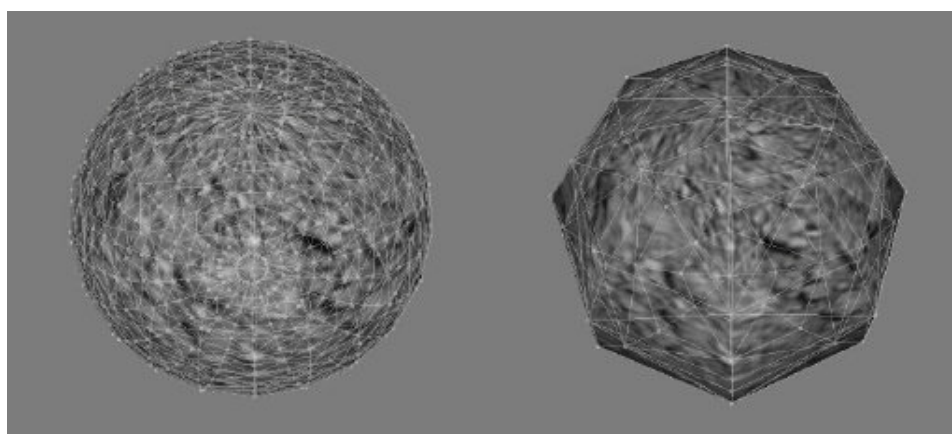


**СОВЕТ**

Для упрощенной сетки используйте только два материала (не текстуры). Первый материал представляет области с полигонами, которые нарисованы на фоне, а второй — скрытые полигоны, применяемые для обнаружения столкновений. Следовательно, во втором материале используется альфа-значение 0.0 (означающее, что он невидим и не будет визуализироваться).

Ранее я упоминал, что вам надо правильно выбрать количество полигонов для визуализации сцены. Если у вас будет слишком много полигонов, движок будет работать медленно; если полигонов слишком мало, во время игры будут возникать графические искажения. Подумайте о следующем: сетка сферы, использующая 500 полигонов явно слишком сложна для использования в качестве упрощенной модели. В упрощенной сетке вам надо использовать достаточное количество полигонов, чтобы представить сферу и гарантировать, что она будет занимать при визуализации точно такую же область экрана.

На рис. 9.7 показана распространенная ошибка создания упрощенной сетки — использование слишком малого количества полигонов.



*Рис. 9.7. У сетки слева много полигонов и она используется при визуализации фона. У упрощенной сетки справа недостаточно полигонов для точного соответствия оригинальной сетке, поэтому трехмерные объекты неправильно наложатся на исключенные пиксели*

**СОВЕТ**

Чтобы сократить количество полигонов в упрощенной сетке, отбросьте те грани, которые никогда не будут видимы и не будут использоваться для проверки столкновений. Также рисуйте только те полигоны, которые действительно могут закрывать трехмерные объекты. Например, если на фоне у вас есть ящик, к которому персонаж игрока никогда не сможет подойти близко, не надо рисовать его в упрощенной сетке. Демонстрационная программа 3Din2D с прилагаемого к книге CD-ROM (находящаяся в папке \BookCode\Chap09\3Din2D) показывает, что использование упрощенных сеток и трехмерных объектов позволяет добиться хороших эффектов, поскольку надо визуализировать только один объект на переднем плане.

Ладно, вы завершили тяжелую работу и создали упрощенную сетку вашей сцены. Упрощенная сетка должна попасть в ваш движок, так что двинемся дальше и преобразуем ее в X-файл. Я сохраняю сетку в файл формата **.3DS** (совместимый со многими программами моделирования), а затем использую утилиту конвертирования от Microsoft (**conv3ds.exe**), поставляемую вместе с DirectX 9 SDK.

#### СОВЕТ

Чтобы использовать **conv3ds.exe** для конвертации из файла **.3DS** в X-файл достаточно ввести в окне интерфейса командной строки следующую команду:

```
conv3ds -mx filename.3ds
```

Показанная команда преобразует сетку из файла **filename.3ds** в **filename.x**. X-файл будет содержать одну сетку (все сетки будут объединены вместе) и записан в текстовом формате. Затем вы можете для экономии места удалить из файла все нормали сетки и данные о наложении текстур (эта информация не используется). Вам нужны только вершины сетки, грани, материалы и список шаблонов материалов сетки.

После преобразования файла **.3DS** (или любого другого файла сетки) в X-файл вы можете использовать классы **cMesh** и **cObject** для быстрой и простой загрузки упрощенной сетки:

```
// Graphics = ранее инициализированный объект cGraphics
cMesh SceneMesh; // Содержит данные сетки
cObject SceneObj; // Используется для визуализации сетки

SceneMesh.Load(&Graphics, "Scene.x");
SceneObj.Create(&Graphics, &SceneMesh);
```

## Визуализация сцены

Вы закончили последний этап, необходимый для того, чтобы изображение фона могло содержать информацию о глубине (с помощью упрощенной сетки). Если вы загрузили фоновое изображение и упрощенную сетку, можно просто визуализировать кадр игры, выполнив следующие действия:

1. Очистить Z-буфер, заполнив его значением 1.0 (и убедиться, что Z-буферизация включена).
2. Визуализировать упрощенную сетку (это заполнит Z-буфер сцены), пропуская полигоны, у которых альфа-значение материала равно 0.0 (это значит, что они невидимы).
3. Отключить Z-буфер.
4. Вывести на экран текстуру с фоном, используя **ID3DXSprite**.
5. Включить Z-буфер.

Поскольку вы уже загрузили фоновое изображение в виде набора из шести текстур, и упрощенная сетка также загружена и готова к

использованию, перейдем к существу и взглянем на код, который визуализирует для вас сцену:

```
// Graphics      = ранее инициализированный объект cGraphics
// Textures[6]   = ранее загруженные текстуры сцены
// SceneObj      = Объект сетки сцены (cObject)

// Очищаем Z-буфер и начинаем сцену
Graphics.ClearZBuffer();
if(Graphics.BeginScene() == TRUE) {

    // Визуализируем упрощенную сетку для установки Z-значений
    Graphics.EnableZBuffer(TRUE);
    SceneObj.Render();

    // Рисуем фон (составленный из шести текстур)
    Graphics.EnableZBuffer(FALSE); // Отключаем Z-буфер
    Graphics.BeginSprite();
    for(long i = 0; i < 2; i++) {
        for(long j = 0; j < 3; j++)
            Textures[i*3+j].Blit(j*256, i*256);
    }
```

Как видите, в последнем фрагменте кода есть два цикла. Они перебирают каждую из шести загруженных вами текстур фона. Каждая текстура копируется на экран с помощью метода **cTexture::Blit**.

---

**ПРИМЕЧАНИЕ**

Большинство современных видеокарт могут работать с текстурами размером до 1024 × 1024 пикселей, а значит, вы можете загрузить в память все фоновое изображение целиком, не деля его на шесть текстур, как я делал в этой главе. Если чувствуете себя достаточно смелым, измените код из этой главы для работы с единой текстурой фона размером 1024 × 1024 пикселей.

---

```
Graphics.EndSprite();

// Завершаем сцену
Graphics.EndScene();
}

// Отображаем сцену
Graphics.Display();
```

---

**ВНИМАНИЕ!**

Перед визуализацией сцены вам надо сделать еще несколько вещей, таких как создание и ориентирование камеры, чтобы она соответствовала камере, использовавшейся при визуализации изображения фона. Вам надо также, чтобы перспективная проекция соответствовала той, которая была задана в программе моделирования при визуализации изображения фона. Если вы используете 3D Studio Max, то угол зрения обычно составляет 34.516 градусов (или 0.6021124 радиан). Также убедитесь в соответствии форматного соотношения, которое обычно равно 1.333333.

---

Вот и все, что надо для фона. Подытожим: сперва вы рисуете упрощенную сетку (для установки в Z-буфере соответствующих сцене значений), а затем заполняете экран изображением фона. Теперь пришло время для самой лучшей части — добавления к сцене трехмерных объектов!

## Добавление трехмерных объектов

После рисования фона ничто не удерживает вас от рисования трехмерных объектов (сеток) в сцене, поскольку Z-буфер содержит требуемые значения глубины для каждого пикселя. Не стесняйтесь. Рисуйте персонажи, объекты и даже дополнения к фоновому изображению. Например, используйте трехмерные объекты для дверей; они могут открываться, закрываться и блокировать перемещение. Ничего не запрещено, наслаждайтесь!

## Столкновения и пересечения

Теперь наступил момент, когда трехмерным объектам надо знать, не сталкиваются ли они с фоном. Если вам надо повторить информацию о том, как определить пересечение сеток, перед тем, как продолжить чтение обратитесь к главе 8. Сейчас вам необходимо гарантировать, что персонажи не будут проходить сквозь стены и препятствия, и определять их высоту для правильного рисования в любой точке сцены.

Если вы смотрите на исходный код упомянутого ранее в этой главе проекта 3Din2D, то поймете, что я позаимствовал код проверки пересечений из главы 8. В программе 3Din2D я блокирую перемещение персонажа сквозь стены и позволяю ему перемещаться вверх и вниз по лестнице. Сейчас я отсылаю вас к этим прекрасным процедурам движения.

## Заканчиваем смешивание графики

Я по-настоящему наслаждался, когда писал эту главу, и надеюсь, вы получили столько же удовольствия читая ее. Настоящий секрет, лежащий в основе поразительных графических движков, таких, как используемый в игре Final Fantasy 7, состоит в том, что никакого секрета нет. Как вы узнали в этой главе, использование представленных в книге простых графических техник позволяет добиться впечатляющих результатов. Держите разум открытым и идеи начнут приходить к вам.

Умелые читатели смогут усовершенствовать программу 3Din2D, добавив к упрощенным сеткам анимацию текстур (с использованием DirectShow, мультимедийного компонента DirectX). Добавление анимации поднимет ваш движок до уровня самых современных игр, таких как Final Fantasy 10.

В примере 2Din3D вы можете использовать разработанный в главе 8 объект **cNodeTreeMesh**, чтобы оптимизировать визуализацию уровня и позволить персонажу свободно перемещаться во всех трех измерениях, а не только вдоль осей X и Y.

### Программы на CD-ROM

Программы, демонстрирующие обсуждавшийся в этой главе код, находятся на прилагаемом к книге CD-ROM. В папке \BookCode\Chap09\ вы найдете следующие программы:

**2Din3D** — демонстрирует использование двумерных объектов в трехмерном мире. Местоположение: \BookCode\Chap09\2Din3D\.

**3Din2D** — показывает как использовать трехмерные объекты в двумерном мире (в стиле игр Final Fantasy). Местоположение: \BookCode\Chap09\3Din2D\.

# Глава 10

## Реализация скриптов

Создавая такой большой проект, как ролевая игра, вы обнаружите, что достаточно сложно (и безрассудно) программировать относящуюся к игре информацию в исходном коде. Гораздо лучше для игровых данных, таких как диалоги, использовать внешние источники (напоминающие программный код), которые называются *скрипты* (*scripts*). В этом случае вы можете контролировать ход вашей игры и экономить время, поскольку не надо повторно компилировать проект, каждый раз, когда вы вносите изменения. В этой главе вы узнаете о том, как создать и использовать простую систему скриптов.

В главе мы рассмотрим следующие темы:

- Что такое скрипты.
- Создание собственной системы скриптов.
- Использование системы скриптов.
- Применение скриптов в играх.

### Что такое скрипты

При создании игры скрипты используются во многом так же, как кинорежиссер использует сценарий — для контроля каждого аспекта вашего «произведения». Игровые скрипты подобны программному коду, который вы пишете, когда создаете игру, за исключением того, что скрипты являются *внешними* по отношению к игровому движку. Поскольку они внешние, вы можете быстро изменить скрипт не компилируя заново весь игровой движок. Вообразите себе проект, в котором больше миллиона строк кода, и вам надо компилировать весь этот проект заново только для того, чтобы изменить одну строку в диалоге!

Со скриптами совсем не трудно работать, и почти каждая часть вашей игры может извлечь пользу от использования скриптов. Вы можете использовать скрипты при навигации по меню, для управления сражениями, для работы с имуществом игрока и многого другого. Представьте себе, что при разработке игры вам захотелось, предоставлять во время битвы игроку список магических заклинаний, которые он регулярно использует для атак. Теперь представьте, что в ходе разработки вы решили изменить некоторые

из этих заклинаний. Если информация о заклинаниях жестко закодирована, вы столкнетесь с серьезной проблемой; вам придется изменить каждый экземпляр программного кода, который управляет заклинаниями, не говоря уже о том, что новый код придется тестировать и отлаживать, чтобы добиться правильной работы. Зачем тратить так много времени на подобные изменения?

Вместо этого вы можете написать код для магических заклинаний и их действий на персонажей игры в нескольких небольших скриптах. Всякий раз, когда начинается сражение, загружаются эти скрипты и отображается панель выбора магических заклинаний. Когда выбирается заклинание, скрипт обрабатывает все эффекты — от наносимых повреждений до перемещений и анимации графических эффектов заклинания.

Для этой книги я колебался между двумя различными типами скриптовых систем. Первая из них предполагает использование языка, похожего на C++. В файле скрипта вы печатаете команды, компилируете его и исполняете скомпилированный скрипт из вашей игры. Вторая система скриптов является значительно упрощенным вариантом первой. Вместо того, чтобы позволить вам печатать команды в файле, эта система обеспечивает создание скриптов путем выбора команд из предварительно определенного набора.

Поскольку моей целью было как можно быстрее предоставить вам работающий скриптовый движок, я выбрал использование второго типа скриптовой системы. Эта система, которую я назвал Mad Lib Scripting, работает используя набор предопределенных команд, называемых *действия* (*actions*), с каждым из которых связана игровая функция. Возьмем, для примера, действия из таблицы 10.1 — у каждого действия есть конкретная выполняемая функция.

**Таблица 10.1.** Примеры команд действий

<b>Действие</b>	<b>Функция</b>
<b>Print</b>	Печатает строку текста на экране.
<b>End</b>	Завершает выполнение скрипта.
<b>Move Character</b>	Перемещает указанный персонаж в заданном направлении.
<b>Play Sound</b>	Воспроизводит указанный звуковой эффект.

С таким ограниченным набором действий вам не нужна мощь сложных компилируемых скриптовых языков; вместо этого вам нужна возможность спросить скриптовую систему, какое действие выполняется, и какие параметры действия должны использоваться при выполнении игровой функции. Самое замечательное в этом методе то, что вместо написания строк кода, определяющих простое действие, вы ссылаетесь на действия и их параметры с помощью чисел.

Для примера скажем, что действию **Play Sound** соответствует номер 4, и действие требует единственного параметра — номера воспроизводимого звука. В скрипте вам потребуется сохранить только два числа: одно для действия, и одно, представляющее звук. Использование чисел для представления действий (вместо текста) делает обработку скриптов такого типа быстрой и простой.

## Создание системы Mad Lib Script

Как я упоминал в предыдущем разделе, я называю рекомендуемую скриптовую систему Mad Lib Script (или, для краткости, MLS), поскольку она воссоздает старую игру на бумаге с тем же именем. В *Mad Libs* (основанной на замечательной концепции для базовой скриптовой системы), вы получаете историю в которой пропущен ряд слов, и ваша задача — заполнить пропуски веселым текстом. Хотя действия в вашей игре и не являются забавными цитатами, сама идея прекрасно подходит для ваших нужд.

В этом разделе я познакомлю вас с концепциями для создания системы Mad Lib Script, от разработки действий, которые будут использоваться в ваших скриптах, до создания системы скриптов (вместе с редактором скриптов), которую вы сможете вставить в собственные игровые проекты.

## Проектирование системы Mad Lib Script

Реализовать собственную систему MLS достаточно просто; создайте действия, которые хотите выполнять в своей игре с «белыми пятнами» (называемыми *подставляемые элементы*, *entries*), которые должны быть заполнены человеком создающим или редактирующим скрипт. Для каждого действия предоставьте список возможных значений для заполнения пустых элементов, которые могут быть разных типов — от строки текста до числа.

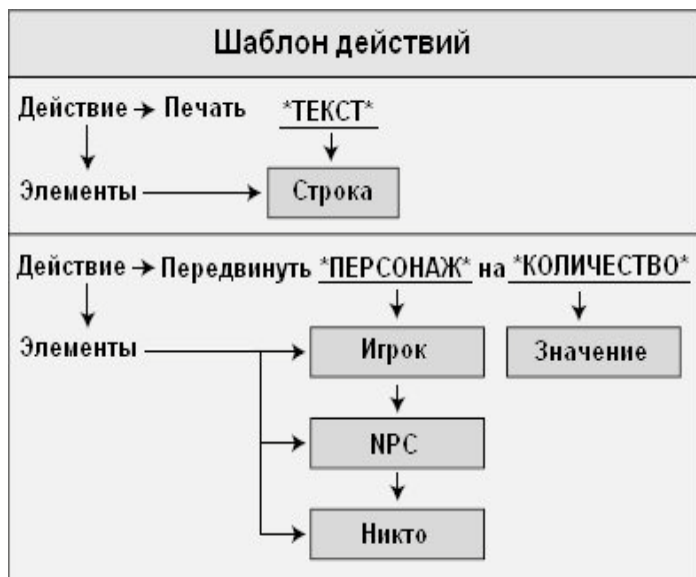
Вы нумеруете действия и пустые элементы, чтобы скриптовая система могла ссылаться на них, как показано в приведенном ниже списке действий:

1. Персонаж (\*ИМЯ\*) получает (\*КОЛИЧЕСТВО\*) повреждений
2. Печатать (\*ТЕКСТ\*)
3. Воспроизвести звуковой эффект (\*ИМЯ\_ЗВУКА\*)
4. Воспроизвести музыку (\*ИМЯ\_МУЗЫКИ\*)
5. Создать объект (\*ИМЯ\_ОБЪЕКТА\*) в координатах (\*XPOS\*), (\*YPOS\*)
6. Завершить обработку скрипта

У каждого из этих шести действий может быть произвольное количество пустых элементов, заключенных в скобки. Каждый из пустых



элементов может хранить строку текста или число. Список действий и возможных значений элементов (с типом элемента) называется *шаблоном действий* (*action template*, пример вы можете увидеть на рис. 10.1).



*Рис. 10.1. Шаблон действий разделяется на несколько действий, которые, в свою очередь, разделяются на элементы*

Как только шаблон действий готов, вы можете ссылаться на действия по их номерам, а не по названиям (которые существуют только для того, чтобы пользователю было легче представить, какую функцию каждое из действий выполняет). Например, теперь я могу сказать, что хочу выполнить действие номер 4, используя `title.mid` в первом пустом элементе. Когда скрипт выполняется, система скриптов видит число 4 (действие номер 4) и знает, что оно требует только один элемент — имя файла с песней, который надо загрузить и воспроизвести.

#### ПРИМЕЧАНИЕ

Скриптовая система MLS будет работать в 90% ваших игр. Например, взгляните на игру *RPG Maker* (от Agetec, Inc) для игровой приставки Sony PlayStation. В ней вы можете создать свою собственную ролевую игру, работая с системой MLS-типа, похожей на ту, что я здесь описываю; поверьте мне, в этой игре вы можете создавать очень сложные скрипты.

Полагаю, вы начали видеть, насколько просто использовать такую систему. Я воздержусь от дальнейшего теоретизирования, чтобы вы могли сразу перейти к программированию собственной системы MLS.

## Программирование системы Mad Lib Script

Чтобы ваша система MLS была максимально мощной, ее надо проектировать таким образом, чтобы она могла поддерживать несколько шаблонов действий, а каждый шаблон действий мог бы содержать неограниченное количество отдельных действий. Благодаря этому, вы сможете использовать систему практически в любом проекте.

Чтобы писать скрипты было проще, используйте программу редактора скриптов (такую, как показана в разделе «Работа с редактором MLS» далее в этой главе) с шаблоном действий, что позволит вам быстро подбирать действия и изменять пустые элементы для них. Когда скрипт создан, вы можете прочитать файл скрипта из вашего движка и обработать каждое отдельное действие, используя указанные для каждого действия подставляемые элементы, которые были введены в редакторе скриптов.

Начнем мы с работы с шаблонами действий.

### ***Работа с шаблонами действий***

Шаблон действий должен содержать список действий, вместе с текстом, количеством подставляемых элементов и данными каждого элемента. Вспомните, что каждое действие нумеруется по его индексу в списке, и так же нумеруется каждый подставляемый элемент в каждом действии. Вы назначаете каждому подставляемому элементу тип (текст, целое число, число с плавающей точкой, логическое значение или выбор варианта). Вы также нумеруете типы, как показано ниже:

1. Тип не задан.
2. Текстовый элемент.
3. Логическое значение.
4. Целое число.
5. Число с плавающей запятой.
6. Выбор варианта (выбор из предопределенного списка текстовых значений).

У каждого типа подставляемого элемента есть собственные уникальные характеристики; строки могут быть переменного размера, числа могут находиться в диапазоне между двумя значениями, а логические значения могут быть **TRUE** или **FALSE**. Что же касается выбора варианта, у каждого варианта есть собственная текстовая строка (скрипт предлагает выбор из списка, а вместо текста используется индекс выбранного варианта).

Обычное действие может выглядеть так:

Действие #1: Произнесение заклинания для (\*ВЫБОР\_ВАРИАНТА\*).

Возможные варианты для элемента #1:

1. Персонаж игрока
2. Произносящий заклинание
3. Цель заклинания
4. Никто

Представьте себе, что вы используете показанное выше действие и говорите, что в качестве цели надо использовать вариант 3. Вы указываете скриптовому движку, что используется действие 1 с вариантом 3 для первого подставляемого элемента (который является выбором варианта). Использование номеров для действий и элементов означает, что обработчику скриптов не надо иметь дело с текстом программы, что значительно упрощает его работу.

Для хранения действий и подставляемых элементов я определил следующие структуры, которые тщательно прокомментированы, чтобы вы могли разобраться в них:

```
// Типы элементов (для пустых полей)
enum Types { _NONE = 0,
             _TEXT,
             _BOOL,
             _INT,
             _FLOAT,
             _CHOICE };

// Структура для хранения информации
// об одном пустом элементе
typedef struct sEntry {
    long Type; // Тип элемента (_TEXT, и т.д.)

    // Следующие два объединения хранят разную информацию
    // об отдельном подставляемом элементе, такую как
    // минимальные и максимальные значения (для целых чисел
    // и чисел с плавающей запятой) и номер варианта
    // для элемента с выбором вариантов. Текстовым и логическим
    // элементам эта информация не нужна.
    union {
        long   NumChoices; // Номер варианта в списке
        long   lMin;       // Минимальное значение для long
        float  fMin;       // Минимальное значение для float
    };

    union {
        long   lMax;       // Максимальное значение для long
        float  fMax;       // Максимальное значение для float
        char   **Choices;  // Массив текстовых описаний для вариантов
    };

    // Конструктор структуры, устанавливающий значения по умолчанию
    sEntry()
    {
        Type      = _NONE;
        NumChoices = 0;
        Choices    = NULL;
    }

    // Деструктор структуры, освобождающий используемые ресурсы
    ~sEntry()
    {
        // Особый случай для выбора варианта
        if(Type == _CHOICE && Choices != NULL) {
            if(NumChoices) {
                for(long i = 0; i < NumChoices; i++) {
                    delete [] Choices[i]; // Удаляем описание варианта
                    Choices[i] = NULL;
                }
            }
            delete [] Choices; // Удаляем массив описаний
            Choices = NULL;
        }
    }
} sEntry;
```

```

// Структура хранит отдельное действие и содержит указатель
// для связанного списка
typedef struct sAction {
    long    ID; // ID действия (от 0 до количества действий минус 1)

    char    Text[256]; // Название действия

    short   NumEntries; // Количество элементов в действии
    sEntry  *Entries;   // Массив структур элементов

    sAction *Next;      // Следующее действие в связанном списке

    sAction()
    {
        ID          = 0; // Устанавливаем значения по умолчанию
        Text[0]      = 0;
        NumEntries   = 0;
        Entries      = NULL;
        Next         = NULL;
    }

    ~sAction()
    {
        delete [] Entries;
        Entries = NULL; // Освобождаем массив элементов
        delete Next;
        Next = NULL;    // Удаляем следующий элемент
    }
} sAction;

```

Вы совместно используете две показанных структуры **sEntry** и **sAction** для хранения описания действия а также типа каждого подставляемого элемента. Типы элементов вы выбираете из предопределенного списка (как было описано ранее в этом разделе). Кроме того, структура **sEntry** содержит правила для каждого типа элемента (используя два объединения).

Поскольку текстовый элемент это только буфер символов, при использовании данного типа нет никаких правил, которым надо следовать. То же самое касается и логических значений, поскольку они могут быть только **TRUE** или **FALSE**. Для целых чисел и чисел с плавающей запятой нужны минимальное и максимальное значения для задания диапазона допустимых значений (отсюда переменные **min/max**). Для выбора варианта необходимо количество вариантов и массив символьных буферов с названиями для каждого варианта.

Структура **sAction** хранит идентификатор действия (номер действия в общем списке действий), текст действия и массив подставляемых элементов, используемых для этого действия. Чтобы определить количество подставляемых элементов в действии (а также тип каждого), необходимо проанализировать текст действия. Для вставки элемента в текст действия используется символ тильды (~), как показано ниже:

Игрок ~ получает ~ очков повреждений

Две тильды представляют два подставляемых элемента. Нам необходимо больше информации о каждом элементе, но как получить ее из двух символов тильда? Это невозможно, и вы должны обратиться к формату хранения шаблона действий, чтобы определить, какая дополнительная информация требуется для каждого из действий.

Шаблон действий хранится в виде текстового файла, в котором текст каждого действия заключен в кавычки. За каждым действием, у которого есть подставляемые элементы (отмеченные в тексте тильдами) следует список параметров элементов. Каждое описание подставляемого элемента начинается со слова, определяющего тип элемента (**ТЕХТ**, **BOOL**, **INT**, **FLOAT** или **CHOICE**). В зависимости от типа элемента, дальше может следовать дополнительная информация.

Для типа **ТЕХТ** дополнительная информация не требуется. То же относится и к типу **BOOL**. Для **INT** и **FLOAT** необходимы минимальное и максимальное значения. И, наконец, за элементом **CHOICE** следует количество вариантов выбора и текст для каждого варианта (заключенный в кавычки).

После того, как каждый подставляемый элемент описан, можно переходить к тексту следующего действия. В приведенном ниже примере шаблона действий продемонстрированы все типы элементов:

```
"Печатать ~"
ТЕХТ
"Переместить персонаж в ~, ~, ~"
FLOAT 0.0 2048.0
FLOAT 0.0 2048.0
FLOAT 0.0 2048.0
"Персонаж ~ ~ ~ очков ~"
CHOICE 3
"Главный персонаж"
"Заклинатель"
"Цель"
CHOICE 2
"Получает"
"Теряет"
INT 0 128
CHOICE 2
"Здоровья"
"Магии"
"Установить переменную ~ в ~"
INT 0 65535
BOOL
"Конец скрипта"
```

Поскольку шаблон действий не позволяет использовать комментарии, я опишу действия и элементы. Первое действие (**Печатать ~**) печатает одну текстовую строку (используя первый элемент в действии, элемент 0). Второе действие получает три значения с плавающей точкой, каждое в диапазоне от 0 до 2048. Третье действие получает три выбираемых значения и одно целое число, которое должно находиться в диапазоне от 0 до 128. В четвертом действии вы также сталкиваетесь с целочисленным значением, а также с

одним логическим значением. Последнее, пятое действие подставляемых элементов не имеет.

Загрузка шаблона действий — это вариация на тему обработки текстового файла с инициализацией соответствующих структур, состоящая из сравнений строк для загружаемых слов и сохранения расположенного внутри кавычек текста. Это действительно простой процесс и в разделе «Совмещаем все вместе в классе `sActionTemplate`» вы точно узнаете, как это сделать.

Следующий шаг — использование шаблона действий совместно с другой структурой, которая хранит данные подставляемого элемента (отображаемый текст, номер выбранного варианта и т.д.), что является назначением элементов скрипта.

### Создание элементов скрипта

Поскольку структура `sEntry` содержит только шаблон (руководство) действия и элементов, вам необходим другой массив структур для хранения данных каждого подставляемого элемента. Эта новая структура включает текст, который используется в текстовом элементе, установленное логическое значение, номер сделанного выбора. Все это содержится в структуре `sScriptEntry`, определение которой выглядит так:

```
typedef struct sScriptEntry
{
    long Type; // Тип элемента (_TEXT, _BOOL, и т.д.)

    union {
        long IOValue; // Используется для сохранения и загрузки
        long Length; // Длина текста (с завершающим 0)
        long Selection; // Выбранный вариант
        BOOL bValue; // Логическое значение
        long lValue; // Целое значение
        float fValue; // Значение с плавающей точкой
    };

    char *Text; // Текстовый буфер

    sScriptEntry()
    {
        Type = _NONE; // Устанавливаем значения по умолчанию
        IOValue = 0;
        Text = NULL;
    }

    ~sScriptEntry()
    {
        delete [] Text;
        Text = NULL;
    } // Удаляем текстовый буфер
} sScriptEntry;
```

Во многом похожая на `sEntry`, `sScriptEntry` хранит действительные значения, используемые в каждом пустом элементе действия. Здесь вы снова видите член `Type`. Он описывает тип элемента

(**\_TEXT**, **\_BOOL** и т.д.). Единственное объединение является самой главной частью; в нем одна переменная для длины текста, одна для номера выбора и по одной для целочисленных значений, значений с плавающей точкой и логических значений.

Следует отметить два момента, относящихся к **sScriptEntry**. Во-первых, указатель на строку символов находится вне объединения (потому что для хранения текстовых данных используются и **Text** и **Length**). Во-вторых, в объединение включена дополнительная переменная **IOValue**. Вы используете **IOValue** для доступа к переменным объединения при сохранении и загрузке данных подставляемого элемента.

Чтобы продемонстрировать, как данные каждого подставляемого элемента хранятся в структуре **sScriptEntry** (или в структурах, если у вас несколько элементов), посмотрите на следующие действия:

```
"~ здоровье игрока на ~"  
CHOICE 2  
"Увеличить"  
"Уменьшить"  
INT 0 65535
```

В зависимости от выбранного в первом элементе варианта, показанное действие увеличивает или уменьшает значение здоровья игрока на заданную величину в диапазоне от 0 до 65 535. Поскольку здесь два подставляемых элемента (выбор варианта и целое число), вам понадобятся две структуры **sScriptEntry**:

```
sScriptEntry Entries;  
  
// Конфигурирование выбора варианта,  
// устанавливаем первый вариант  
Entries[0].Type = _CHOICE;  
Entries[0].Selection = 0; // Увеличение  
  
// Конфигурирование целочисленного значения,  
// устанавливаем 128  
Entries[1].Type = _INT;  
Entries[1].lValue = 128;
```

Когда мы имеем дело с элементами скрипта, возникает достаточно сложная задача, если в завершенном скрипте много элементов. Каждому действию скрипта необходима соответствующая структура **sEntry**, которая, в свою очередь, может содержать несколько структур **sScriptEntry**. Прежде чем думая об этом вы окажетесь по колено в структурах, поговорим о беспорядке! Для лучшей обработки структур скрипта нам нужна другая структура, которая отслеживает каждый элемент, принадлежащий действию скрипта.

```
typedef struct sScript  
{  
    long Type; // от 0 до (количество действий - 1)  
    long NumEntries; // Количество элементов в данном действии скрипта
```

```

sScriptEntry *Entries; // Массив подставляемых элементов

sScript *Prev; // Предыдущий элемент связанного списка
sScript *Next; // Следующий элемент связанного списка

sScript()
{
    Type = 0; // Установка значений по умолчанию
    NumEntries = 0;
    Entries = NULL;
    Prev = Next = NULL;
}

~sScript()
{
    delete [] Entries;
    Entries = NULL; // Удаление массива элементов

    delete Next; // Удаление следующего элемента
    Next = NULL; // связанного списка
}
} sScript;

```

Вы используете структуру **sScript** для хранения отдельного действия, а также для управления связанным списком других структур **sScript**, образующих скрипт в целом. Значение переменной **Type** может находиться в диапазоне от 0 до числа на единицу меньшего, чем количество действий в шаблоне действий. Если в вашем шаблоне действий десять действий, **Type** может принимать значения от нуля до девяти.

Для упрощения обработки в переменной **NumEntries** хранится количество элементов. Значение **NumEntries** должно соответствовать значению переменной с количеством элементов в шаблоне действий. Затем выделяется память для массива структур **sScriptEntry**, которые будут хранить реальные данные для каждого элемента из шаблона действий. Если с данным действием связаны два подставляемых элемента, то вам необходимо выделить память под две структуры **sScriptEntry**.

И, наконец, два указателя в **sScript** — **Prev** и **Next**. Эти два указателя управляют связанным списком, образующим скрипт. Для создания связанного списка структур **sScript** (такого, как показанный на рис. 10.2), начните с корневой структуры, представляющей первое действие скрипта. Затем свяжите структуры **sScript** через переменные **Next** и **Prev**, как показано ниже:

```

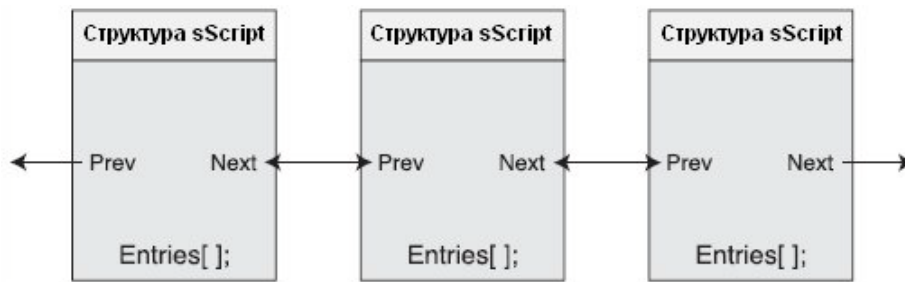
sScript *ScriptRoot = new sScript();
sScript *ScriptPtr = new sScript();

ScriptRoot->Next = ScriptPtr; // Указатель на второе действие

ScriptPtr->Prev = ScriptRoot; // Указатель обратно на корень

```





**Рис. 10.2.** Связанный список действий скрипта использует переменные *Prev* и *Next* для объединения скрипта в единое целое. У каждого действия скрипта есть собственный массив элементов

Теперь вы можете начать с корня скрипта и пройти вниз по всему скрипту с помощью следующего кода:

```
void TraverseScript(sScript *pScript)
{
    // Цикл, пока в скрипте больше не останется действий
    while(pScript != NULL) {

        // Делаем что-нибудь с pScript
        // pScript->Type хранит ID действия скрипта

        // Переходим к следующему действию скрипта
        pScript = pScript->Next;
    }
}
```

Используя связанный список вы можете быстро загрузить и сохранить скрипт, что демонстрируют следующие две функции:

```
BOOL SaveScript(char *Filename, sScript *ScriptRoot)
{
    FILE *fp;
    long i, j, NumActions;
    char Text[256];
    sScript *ScriptPtr;

    // Убеждаемся, что в скрипте есть действия
    if((ScriptPtr = ScriptRoot) == NULL)
        return FALSE;

    // Подсчет количества действий
    NumActions = 0;
    while(ScriptPtr != NULL) {
        NumActions++;
        ScriptPtr = ScriptPtr->Next; // Следующее действие
    }
}
```

В показанном выше фрагменте кода вы подсчитываете количество действий в скрипте. Теперь вам надо открыть выходной файл, записать количество сохраняемых действий, и в цикле перебрать каждое действие, записывая относящиеся к нему данные.

```
// Открываем выходной файл
if((fp = fopen(Filename, "wb")) == NULL)
    return FALSE; // Сообщаем об ошибке
```

```

// Выводим количество действий в скрипте
fwrite(&NumActions, 1, sizeof(long), fp);

// Перебираем каждое действие скрипта
ScriptPtr = ScriptRoot;
for(i = 0; i < NumActions; i++) {
    // Выводим тип действия и количество элементов в нем
    fwrite(&ScriptPtr->Type, 1, sizeof(long), fp);
    fwrite(&ScriptPtr->NumEntries, 1, sizeof(long), fp);

    // Выводим данные подставляемых элементов (если они есть)
    if(ScriptPtr->NumEntries) {
        for(j = 0; j < ScriptPtr->NumEntries; j++) {
            // Записываем тип элемента и данные
            fwrite(&ScriptPtr->Entries[j].Type, 1,
                sizeof(long), fp);
            fwrite(&ScriptPtr->Entries[j].IOValue, 1,
                sizeof(long), fp);

            // Записываем текст элемента (если есть)
            if(ScriptPtr->Entries[j].Type == _TEXT &&
                ScriptPtr->Entries[j].Text != NULL)
                fwrite(ScriptPtr->Entries[j].Text, 1,
                    ScriptPtr->Entries[j].Length, fp);
        }
    }
    // Переходим к следующей структуре в связанном списке
    ScriptPtr = ScriptPtr->Next;
}
fclose(fp);
return TRUE; // Сообщаем об успехе!
}

```

Что касается загрузки, вы следуете тому же самому шаблону действий, что и при сохранении скрипта, только загружаете данные действий, вместо того, чтобы записывать их, — откройте файл для чтения, прочитайте количество действий и прочитайте данные каждого действия.

```

sScript *LoadScript(char *Filename, long *NumActions)
{
    FILE *fp;
    long i, j, Num;
    char Text[2048];

    sScript *ScriptRoot, *Script, *ScriptPtr = NULL;

    // Открываем файл для ввода
    if((fp = fopen(Filename, "rb")) == NULL)
        return NULL;

    // Получаем количество действий в файле скрипта
    fread(&Num, 1, sizeof(long), fp);

    // Сохраняем количество действий в предоставленной
    // пользователем переменной
    if(NumActions != NULL) *NumActions = Num;

    // Цикл по каждому действию скрипта
    for(i = 0; i < Num; i++) {
        // Выделяем память для структуры скрипта и привязываем ее
        Script = new sScript();
    }
}

```

```
if (ScriptPtr == NULL)
    ScriptRoot = Script; // Назначаем корень
else
    ScriptPtr->Next = Script;

ScriptPtr = Script;

// Получаем тип действия и количество элементов
fread(&Script->Type, 1, sizeof(long), fp);
fread(&Script->NumEntries, 1, sizeof(long), fp);

// Получаем данные элементов (если они есть)
if (Script->NumEntries) {
    // Выделяем память для массива элементов
    Script->Entries = new sScriptEntry[Script->NumEntries]();

    // Загружаем каждый элемент
    for (j = 0; j < Script->NumEntries; j++) {
        // Получаем тип элемента и данные
        fread(&Script->Entries[j].Type, 1, sizeof(long), fp);
        fread(&Script->Entries[j].IOValue, 1, sizeof(long), fp);

        // Получаем текст (если есть)
        if (Script->Entries[j].Type == _TEXT &&
            Script->Entries[j].Length) {
            // Выделяем буфер и получаем строку
            Script->Entries[j].Text =
                new char[Script->Entries[j].Length];
            fread(Script->Entries[j].Text, 1,
                Script->Entries[j].Length, fp);
        }
    }
}
fclose(fp);
return ScriptRoot;
}
```

Получив корневую структуру скрипта из связанного списка **SaveScript** выводит данные каждой структуры скрипта, которые включают номер действия, количество следующих далее подставляемых элементов, данные элементов и необязательный текст для текстового элемента. Весь связанный список структур **sScript** записывается в файл.

Функция **LoadScript** открывает файл скрипта и строит связанный список структур **sScript** на основании загруженных данных. Образующие связанный список структуры **sScriptEntry** и **sScript** создаются на лету. После завершения работы функция **LoadScript** инициализирует переменную **NumActions**, записывая в нее количество загруженных действий, и возвращает указатель на корневую структуру скрипта.

### **Объединяем все в классе *cActionTemplate***

Вы познакомились со структурами, используемыми для шаблонов действий и для хранения данных скрипта. Теперь пришло время объединить все части вместе и создать работающий класс, который будет загружать и обрабатывать скрипты:

```

class cActionTemplate
{
private:
    long    m_NumActions;    // Количество действий в шаблоне
    sAction *m_ActionParent; // Список действий шаблона

    // Функции чтения текста (в основном используются в действиях)
    BOOL GetNextQuotedLine(char *Data, FILE *fp, long MaxSize);
    BOOL GetNextWord(char *Data, FILE *fp, long MaxSize);

public:
    cActionTemplate();
    ~cActionTemplate();

    // Загрузка и освобождение шаблона действий
    BOOL Load(char *Filename);
    BOOL Free();

    // Получение количества действий в шаблоне,
    // родительского действия и заданной структуры действия
    long GetNumActions();
    sAction *GetActionParent();
    sAction *GetAction(long Num);

    // Получение структуры sScript заданного типа
    sScript *CreateScriptAction(long Type);

    // Получение информации о действиях и элементах
    long GetNumEntries(long ActionNum);
    sEntry *GetEntry(long ActionNum, long EntryNum);

    // Формирование текста действия с использованием
    // значений по умолчанию
    BOOL ExpandDefaultActionText(char *Buffer, sAction *Action);

    // Формирование текста действия с использованием
    // установленных значений
    BOOL ExpandActionText(char *Buffer, sScript *Script);
};

```

В коде есть две функции, с которыми мы еще не встречались в этой главе, — **GetNextQuotedLine** и **GetNextWord**. Функция **GetNextQuotedLine** сканирует файл, выбирая заключенную в кавычки строку текста, а функция **GetNextWord** читает из файла очередное слово. Обе функции получают указатель на буфер данных, в котором будут сохранять текст, указатель для доступа к файлу и максимальный размер буфера данных (для предотвращения переполнения).

```

BOOL cActionTemplate::GetNextQuotedLine(char *Data,
                                         FILE *fp, long MaxSize)
{
    int    c;
    long Pos = 0;

    // Читаем, пока не найдем кавычки (или EOF)
    while(1) {
        if((c = fgetc(fp)) == EOF)
            return FALSE;

        if(c == '"') {

```

Здесь мы просто читаем символ (байт) и проверяем, не является ли он кавычкой. Если это кавычка, мы входим в цикл, который строит строку, последовательно читая символы, пока не будет достигнута следующая кавычка.

```
// Читаем, пока не достигнем следующей кавычки (или EOF)
while(1) {
    if((c = fgetc(fp)) == EOF)
        return FALSE;

    // Обнаружив вторую кавычку возвращаем текст
    if(c == '"') {
        Data[Pos] = 0;
        return TRUE;
    }

    // Добавляем допустимые символы к строке
    if(c != 0x0a && c != 0x0d) {
        if(Pos < MaxSize - 1)
            Data[Pos++] = c;
    }
}
}
```

Функция **GetNextWord** работает почти так же, как **GetNextQuotedLine** — она последовательно считывает символы в буфер, пока не будет обнаружен конец строки или пробел (оба отмечают конец загружаемого слова).

```
BOOL cActionTemplate::GetNextWord(char *Data, FILE *fp, long MaxSize)
{
    int c;
    long Pos = 0;

    // Очищаем строку
    Data[0] = 0;

    // Читаем, пока не найдем допустимый символ
    while(1) {
        if((c = fgetc(fp)) == EOF) {
            Data[0] = 0;
            return FALSE;
        }

        // Проверка на начало слова
        if(c != 32 && c != 0x0a && c != 0x0d) {
            Data[Pos++] = c;

            // Цикл, пока не достигнем конца слова (или EOF)
            while((c = fgetc(fp)) != EOF) {
                // Прерываемся на разделителе слов
                if(c == 32 || c == 0x0a || c == 0x0d)
                    break;

                // Добавляем символ, если есть место
                if(Pos < MaxSize - 1)
                    Data[Pos++] = c;
            }
        }
    }
}
```

```

    }
    // Добавляем символ окончания строки
    Data[Pos] = 0;
    return TRUE;
}
}
}

```

Используя функции **GetNextQuotedLine** и **GetNextWord** вы можете просканировать содержащий описание действий текст входного файла, что и делает функция **cActionTemplate::Load**:

```

BOOL cActionTemplate::Load(char *Filename)
{
    FILE *fp;
    char Text[2048];

    sAction *Action, *ActionPtr = NULL;
    sEntry *Entry;

    long i, j;

    // Освобождаем предыдущие структуры действий
    Free();

    // Открываем файл действий
    if((fp = fopen(Filename, "rb")) == NULL)
        return FALSE;

    // Цикл, пока не достигнем конца файла
    while(1) {

```

Здесь вы открыли файл шаблона действий и вошли в цикл. В теле цикла вы читаете строки текста в кавычках (которые содержат шаблоны действий) и анализируете их, определяя какие данные должны быть загружены дальше для описания действия:

```

        // Получаем следующее действие в кавычках
        if(GetNextQuotedLine(Text, fp, 2048) == FALSE)
            break;

        // Выход, если нет текста действия
        if(!Text[0])
            break;

        // Создаем структуру действия и присоединяем
        // ее к списку
        Action = new sAction();
        Action->Next = NULL;

        if(ActionPtr == NULL)
            m_ActionParent = Action;
        else
            ActionPtr->Next = Action;

        ActionPtr = Action;

        // Копируем текст действия
        strcpy(Action->Text, Text);

```

```
// Сохраняем ID действия
Action->ID = m_NumActions;

// Увеличиваем счетчик загруженных действий
m_NumActions++;

// Подсчитываем количество элементов в действии
for(i = 0; i < (long)strlen(Text); i++) {
    if(Text[i] == '~')
        Action->NumEntries++;
}
```

После выделения памяти под структуру шаблона действия вы присоединяете ее к списку шаблонов действий, сохраняете в ней текст действия и подсчитываете количество элементов в тексте действия. Каждый элемент отмечается знаком тильды (~) и для каждого символа тильды в файле должна быть соответствующая строка текста, описывающая тип используемого элемента. Представленный ниже код обрабатывает загрузку данных элементов:

```
// Выделяем память и читаем элементы (если есть)
if(Action->NumEntries) {
    Action->Entries = new sEntry[Action->NumEntries]();
    for(i = 0; i < Action->NumEntries; i++) {
        Entry = &Action->Entries[i];

        // Получаем тип элемента
        GetNextWord(Text, fp, 2048);

        // Тип TEXT, нет последующих данных
        if(!strcmp(Text, "TEXT")) {
```

В показанном фрагменте кода вы проверяете совпадает ли первое слово в описании элемента со строкой «TEXT». Если да, значит тип данного элемента **TEXT** и вы должны получить текстовую строку в качестве данных элемента. Последующий код проверяет другие типы элементов (**INT**, **BOOL** и т.д.) и загружает последующие данные, согласно объявленному типу элемента:

```
        // Устанавливаем текстовый тип
        Entry->Type = _TEXT;
    } else // Если не TEXT, проверяем тип INT
        // Тип INT, получаем минимальное
        // и максимальное значение
        if(!strcmp(Text, "INT")) {
            // Устанавливаем тип INT и создаем элемент INT
            Entry->Type = _INT;

            // Получаем минимальное значение
            GetNextWord(Text, fp, 2048);
            Entry->lMin = atol(Text);

            // Получаем максимальное значение
            GetNextWord(Text, fp, 2048);
            Entry->lMax = atol(Text);
        } else // Если не INT, проверяем тип FLOAT
            // Тип FLOAT, получаем минимальное и
            // максимальное значения
```

```

        if(!strcmp(Text, "FLOAT")) {
            // Устанавливаем тип FLOAT и создаем
            // элемент FLOAT
            Entry->Type = _FLOAT;

            // Получаем минимальное значение
            GetNextWord(Text, fp, 2048);
            Entry->fMin = (float)atof(Text);

            // Получаем максимальное значение
            GetNextWord(Text, fp, 2048);
            Entry->fMax = (float)atof(Text);
        } else // Если не FLOAT, проверяем тип BOOL
        // Тип BOOL, нет параметров
        if(!strcmp(Text, "BOOL")) {
            // Устанавливаем тип BOOL и создаем элемент BOOL
            Entry->Type = _BOOL;
        } else // Если не BOOL, проверяем тип CHOICE
        // Тип CHOICE, получаем количество вариантов
        // и их текст
        if(!strcmp(Text, "CHOICE")) {
            // Устанавливаем тип CHOICE и создаем элемент CHOICE
            Entry->Type = _CHOICE;

            // Получаем количество вариантов
            GetNextWord(Text, fp, 1024);
            Entry->NumChoices = atol(Text);
            Entry->Choices = new char[Entry->NumChoices];

            // Получаем текст каждого варианта
            for(j = 0; j < Entry->NumChoices; j++) {
                GetNextQuotedLine(Text, fp, 2048);
                Entry->Choices[j] = new char[strlen(Text) + 1];
                strcpy(Entry->Choices[j], Text);
            }
        }
    }
}
fclose(fp);
return TRUE;
}

```

Используя функцию **cActionTemplate::Load** вы открываете текстовый файл и начинаете сканировать его. В начале каждой итерации в новую структуру **sAction** загружается очередная строка в кавычках (действие), которая затем проверяется на наличие символов тильда. Если символы тильда найдены, загружается и разбирается остальная информация. Этот процесс продолжается пока не будет достигнут конец файла.

Перейдем к следующей функции в **cActionTemplate** — **CreateScriptAction**; она получает номер действия и возвращает инициализированную структуру **sScript**, подготовленную для хранения необходимого действию количества элементов. Затем вы можете непосредственно разбирать структуру **sScript** для доступа к данным, находящимся в действии и элементах (что делает редактор MLS и примеры):



```
sScript *cActionTemplate::CreateScriptAction(long Type)
{
    long i;

    sScript *Script;
    sAction *ActionPtr;

    // Проверяем, что это допустимое действие - тип
    // должен быть реально существующим ID действия
    // (из уже загруженного списка действий).
    if(Type >= m_NumActions)
        return NULL;

    // Получаем указатель на действие
    if((ActionPtr = GetAction(Type)) == NULL)
        return NULL;

    // Создаем новую структуру sScript
    Script = new sScript();

    // Устанавливаем тип и количество элементов
    // (выделяем список)
    Script->Type = Type;
    Script->NumEntries = ActionPtr->NumEntries;
    Script->Entries = new sScriptEntry[Script->NumEntries]();

    // Инициализируем каждый элемент
    for(i = 0; i < Script->NumEntries; i++) {
        // Сохраняем тип
        Script->Entries[i].Type = ActionPtr->Entries[i].Type;

        // Инициализируем данные элемента в зависимости от типа
        switch(Script->Entries[i].Type) {
            case _TEXT:
                Script->Entries[i].Text = NULL;
                break;
            case _INT:
                Script->Entries[i].lValue = ActionPtr->Entries[i].lMin;
                break;
            case _FLOAT:
                Script->Entries[i].fValue = ActionPtr->Entries[i].fMin;
                break;
            case _BOOL:
                Script->Entries[i].bValue = TRUE;
                break;
            case _CHOICE:
                Script->Entries[i].Selection = 0;
                break;
        }
    }
    return Script;
}
```

Последние две функции в **cActionTemplate** это **ExpandDefaultActionText** и **ExpandActionText**. Обе функции берут текст действия и заменяют символы тильда внутри него на более понятный текст, такой как целые числа или описание варианта выбора.

Различие между функциями в том, что **ExpandDefaultActionText** формирует текст не зависящий от данных элементов; она просто

берет минимальное значение или первый вариант выбора. **ExpandActionText** формирует текст действия используя данные, содержащиеся в предоставляемой структуре **sScript**. Обе функции используются только в редакторе скриптов, чтобы сделать более удобными для просмотра данные, содержащиеся в шаблоне действий и структурах скрипта. Код функций вы можете посмотреть в программах на CD-ROM (в проекте MLS Script Editor).

---

**ПРИМЕЧАНИЕ**

Я не включил функции сохранения и загрузки скриптов, поскольку они не являются частью шаблона действий. Однако, вы можете модифицировать функции сохранения и загрузки для каждого приложения, как сочтете нужным. Это справедливо и для двух примеров программ к этой главе, MlsEdit и MlsDemo, которые находятся на прилагаемом к книге CD-ROM (обе программы расположены в каталоге \BookCode\Chap10).

---

Разобравшись с шаблонами действий и структурами скриптов вы можете начать объединять их вместе и применять MLS, а начинается все это с редактора скриптов Mad Lib.

## Работа с редактором MLS Editor

Система MLS работает только с числами: числа представляют действия, затем следует количество элементов и числа же представляют данные элементов. Компьютеры замечательно работают с числами, но человеку требуется что-то более удобное. Желательно создавать скрипты из понятных строк текста, а преобразованием текста в набор числовых представлений, которые может обрабатывать система скриптов, пусть занимается редактор.

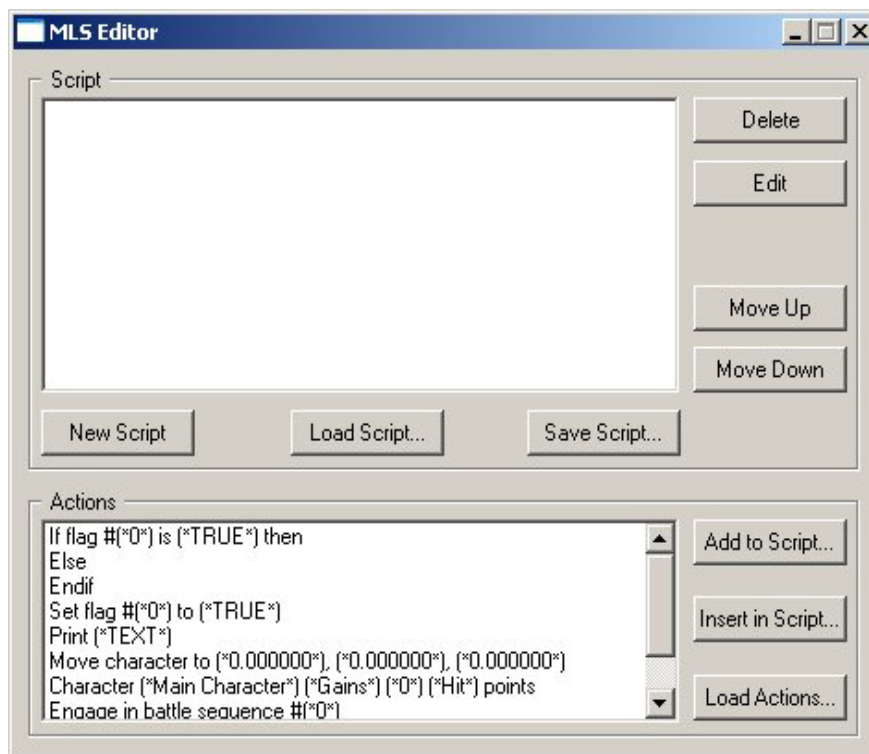
При редактировании скрипта работа с числами тоже нам не подходит, так что у редактора есть задача загрузить эти числа и преобразовать обратно в строки текста, которые можно легко прочитать. Итак, чтобы все стало ясно, нам необходимо конструировать скрипты, используя набор текстовых команд и позволить редактору скриптов и движку конвертировать эти команды в их числовое представление и наоборот.

Редактор скриптов Mad Lib импортирует текст, представляющий действия и предоставляет пользователю возможность редактировать список действий и модифицировать пустые элементы в каждом действии. На рис. 10.3 показан редактор MLS Editor, который я создал для этой книги.

Текстовое поле, содержащее редактируемый в данный момент скрипт, находится в верхней части окна приложения. Под ним расположено поле, где перечислены все действия из загруженного шаблона действий. Также в окне есть различные кнопки, используемые для конструирования скрипта.

Вы найдете, что использование редактора скриптов интуитивно понятно. У вас есть команды для загрузки набора действий, загрузки и сохранения скрипта, создания нового скрипта, добавления, удаления и

изменения строк скрипта, а также для перемещения строк в скрипте вверх или вниз. Используемые редактором действия хранятся в файле шаблона действий.



*Рис. 10.3. Редактор MLS Editor содержит все необходимое для создания и редактирования скриптов*

Что касается действительных элементов скрипта, редактор использует структуры **sScript** и **sScriptEntry** для хранения редактируемого в данный момент скрипта, сохранения и загрузки, точно так же, как мы видели раньше.

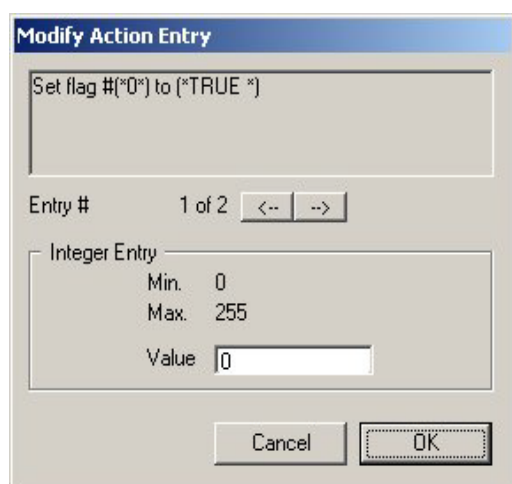
Чтобы начать сеанс редактирования MLS загрузите шаблон действий или используйте предлагаемый мной по умолчанию шаблон, который находится в файле **default.mla** (вы найдете его в каталоге **\BookCode\Chap10\Data**). Затем вы можете добавлять, вставлять и редактировать строки скрипта, используя соответствующие кнопки в окне редактора. Описание всех кнопок приведено в таблице 10.2.

Начав добавлять действия к скрипту (с помощью кнопок **Add to Script** или **Insert in Script**), обратите внимание, что текст действия отображается в развернутом виде и добавляется к тексту в поле скрипта (текстовое поле редактирования скрипта расположено в верхней части окна редактора скриптов). Действия скрипта сохраняются в порядке сверху вниз, и корнем, с которого начинается скрипт, является самое верхнее действие. Обработка скрипта начинается сверху и идет вниз, также как выполнение кода C/C++.

Обратите внимание, что каждый раз, когда вы добавляете, вставляете или редактируете строку скрипта открывается диалоговое окно **Modify Action Entry** (рис. 10.4). Вы используете это окно для модификации подставляемых элементов действия скрипта.

Таблица 10.2. Кнопки редактора MLS Editor

Кнопка	Функция
<b>Delete</b>	Удаляет выбранную строку из текста скрипта.
<b>Edit</b>	Редактирует элементы выбранной строки скрипта.
<b>Move Up</b>	Перемещает выбранное действие скрипта вверх.
<b>Move Down</b>	Перемещает выбранное действие скрипта вниз.
<b>New Script</b>	Удаляет все действия скрипта из памяти и начинает работу с чистого листа.
<b>Load Script</b>	Загружает файл скрипта с диска (файл с расширением .MLS).
<b>Save Script</b>	Сохраняет скрипт в файле на диске (файл с расширением .MLS).
<b>Add to Script</b>	Добавляет выбранное действие (из списка действий) в конец скрипта. Автоматически открывается диалоговое окно <b>Modify Action Entry</b> .
<b>Insert in Script</b>	Вставляет выбранное действие (из списка действий) в отмеченную строку скрипта. Автоматически открывается диалоговое окно <b>Modify Action Entry</b> .
<b>Load Actions</b>	Загружает новый файл шаблона действий (файл с расширением .MLA). Также принудительно очищает весь текст скрипта.



*Рис. 10.4. Используйте диалоговое окно **Modify Action Entry** для быстрой навигации и модификации элементов действий скрипта*

В диалоговом окне **Modify Action Entry** расположены различные элементы управления для модификации элементов действий скрипта. Вам предоставляется два текстовых поля. В первом (находящемся в диалоговом окне сверху) вам выводится текст элемента или минимальное и максимальное значения для диапазона; во втором вы вводите значения, относящиеся к элементу. У логических значений есть два переключателя: один для выбора значения **TRUE**, а другой — для **FALSE**. Для элемента с

выбором вариантов в диалоговом окне отображается список доступных вариантов.

В диалоговом окне **Modify Action Entry** есть несколько элементов управления общих для всех типов элементов. Во-первых, поле в котором отображается текст действия (с преобразованными в текст элементами). Затем, в поле **Entry #** выводится номер редактируемого в данный момент элемента и общее количество элементов в действии. Для навигации по элементам используются две кнопки — переход к предыдущему элементу (на кнопке изображена стрелка, направленная влево) и переход к следующему элементу (на кнопке изображена стрелка, направленная вправо). Щелчок по любой из этих кнопок приводит к обновлению данных текущего подставляемого элемента и выводу данных следующего.

В нижней части диалогового окна **Modify Action Entry** есть еще две кнопки — **OK** и **Cancel**. Кнопка **Cancel** отображается только когда вы добавляете действие. Когда вы выбираете действие для редактирования из текста скрипта, кнопка **Cancel** не отображается, а значит все сделанные изменения сохраняются сразу, независимо от того, щелкнули вы по кнопке **OK** или нет, так что будьте внимательны, чтобы не модифицировать что-нибудь не требующее вашего вмешательства. Щелчок по кнопке **OK** сохраняет все данные элементов и добавляет, вставляет или модифицирует действие, выбранное в диалоговом окне **MLS Editor**.

Вместе с редактором скриптов на компакт-диске записаны пример шаблона действий и скрипта, чтобы помочь начать работать. Настоящую мощь вы почувствуете когда начнете конструировать собственные шаблоны действий, приспособленные к вашим игровым проектам. После того, как вы создадите шаблон действий и сконструируете свой скрипт, можно сосредоточиться на их использовании в своем проекте.

---

**ПРИМЕЧАНИЕ**

Код редактора **MLS** находится на прилагаемом к книге **CD-ROM** в папке `\BookCode\Chap10\MLSEdit`.

---

## Выполнение скриптов **Mad Lib**

Уф! Могу честно сказать, что самая сложная часть позади, поскольку реализовать сейчас выполнение скриптов — это просто детские игрушки. Вы можете выбросить за дверь шаблоны действий, потому что теперь будете работать только со структурами **sScript** и **sScriptEntry**.

Первый этап работы со скриптом — загрузка его в память, выполняемая с помощью функции **LoadScript** (если вы хотите больше узнать об этой функции, обратитесь к разделу «Создание элементов скрипта»).

```
long NumActions;  
sScript *LoadedScript = LoadScript("Script.mls", &NumActions);
```

После этого ваш игровой движок просто перебирает элементы связанного списка скрипта для выполнения каждого действия. Это требует

некоторого объема кодирования, поскольку в этом случае действия известны только по номерам (так что вы должны знать, что каждое из действий делает). Вот пример, перебирающий элементы загруженного скрипта и ищущий действие **Print** (действие 0), содержащее единственный элемент (текст для печати):

```
sScript *ScriptPtr = LoadedScript; // Начинаем с корня

// Перебираем в цикле все действия скрипта
while(ScriptPtr != NULL) {
    // Это действие 0?
    if(ScriptPtr->Type == 0) {
        // У данного действия один элемент - текст.
        // Отображаем текст в окне сообщений
        MessageBox(NULL, ScriptPtr->Entries[0].Text, "ТЕХТ", MB_OK);
    }
    // Переходим к следующему действию в скрипте
    ScriptPtr = ScriptPtr->Next;
}
```

Хотя здесь всего несколько строк кода, они демонстрируют удивительный потенциал обработки скриптов. Немного изобретательности и вы сможете приспособить MLS для обработки большинства скриптовых действий.

Как насчет использования условных инструкций **if...then...else**? Как вы знаете, эта инструкция проверяет, равно значение условия **true** или **false**, и, в зависимости от результата, выполняет различные последовательности действий. Возьмем, к примеру, следующий код на C:

```
BOOL GameFlags[256]; // Флаги, используемые в игре

if(GameFlags[0] == TRUE) {
    // Печатаем сообщение и устанавливаем флаг в FALSE
    MessageBox(NULL, "It's TRUE!", "Message", MB_OK);
    GameFlags[0] = FALSE;
} else {
    // Печатаем сообщение
    MessageBox(NULL, "It's FALSE.", "Message", MB_OK);
}
```

В зависимости от значений, хранящихся в массиве **GameFlags**, выполняются различные блоки кода. Создав несколько действий и слегка переработав код обработки скрипта, вы можете наслаждаться преимуществами использования конструкции **if...then...else** в MLS. Сперва взгляните на шаблон действий:

```
"If GameFlag ~ equals ~ then"
INT 0 255
BOOL
"Else"
"EndIf"
"Set GameFlag ~ to ~"
INT 0 255
BOOL
"Print ~"
TEXT
```

Здесь нет ничего особенного, поскольку настоящая работа происходит в коде выполнения скрипта:

```
// pScript = загруженный скрипт, содержащий следующий код:
// "If GameFlag (0) equals (TRUE) then"
// "Print (It's TRUE!)"
// "Set GameFlag (0) to (FALSE)"
// "Else"
// "Print (It's FALSE.)"
// "EndIf"

// Функции обработки действий
sScript *Script_IfThen(sScript *Script);
sScript *Script_Else(sScript *Script);
sScript *Script_EndIf(sScript *Script);
sScript *Script_SetFlag(sScript *Script);
sScript *Script_Print(sScript *Script);

// Структура исполнения действия скрипта
typedef struct sScriptProcesses {
    sScript *(*Func)(sScript *ScriptPtr);
} sScriptProcesses;

// Список структур исполнения действий скрипта
sScriptProcesses ScriptProcesses[] = {
    { Script_IfThen },
    { Script_Else },
    { Script_EndIf },
    { Script_SetFlag },
    { Script_Print }
}

BOOL GameFlags[256]; // Массив игровых флагов

void RunScript(sScript *pScript)
{
    // Очищаем массив GameFlags,
    // устанавливая везде значение FALSE
    for(short i = 0; i < 256; i++)
        GameFlags[i] = FALSE;

    // Сканируем скрипт и выполняем функции
    while(pScript != NULL) {
        // Вызываем функцию скрипта и прерываем обработку,
        // если она вернула NULL. Любое другое
        // возвращенное значение - это указатель на следующую
        // функцию, обычно pScript->Next.
        pScript = ScriptProcesses[pScript->Type].Func(pScript);
    }
}

sScript *Script_IfThen(sScript *Script)
{
    BOOL Skipping; // Флаг пропуска действий скрипта

    // Смотрим, совпадает ли флаг со вторым элементом
    if(g_Flags[Script->Entries[0].lValue % 256] ==
        Script->Entries[1].bValue)
        Skipping = FALSE;
    else
        Skipping = TRUE;
```

```

// Здесь флаг Skipping установлен, если действия скрипта должны быть
// пропущены согласно условной инструкции if ... then. Действия
// выполняются, если Skipping = FALSE, также ищется else для
// переключения режима пропуска или endif для завершения
// условного блока

// Переходим к обработке следующего действия
Script = Script->Next;

while(Script != NULL) {
    // Если Else, переключаем режим пропуска
    if(Script->Type == 1)
        Skipping = (Skipping == TRUE) ? FALSE : TRUE;

    // Прерывание на EndIf
    if(Script->Type == 2)
        return Script->Next;

    // Обрабатываем функции скрипта в условном блоке,
    // при этом действия пропускаются, если условие не выполнено
    if(Skippping == TRUE)
        Script = Script->Next;
    else {
        if((Script =
            ScriptProcesses[Script->Type].Func(Script)) == NULL)
            return NULL;
    }
}
return NULL; // Достигнут конец скрипта
}

sScript *Script_SetFlag(sScript *Script)
{
    // Установка логического флага
    GameFlags[Script->Entries[0].lValue % 256] =
        Script->Entries[1].bValue;
}

sScript *Script_Else(sScript *Script) { return Script->Next; }

sScript *Script_EndIf(sScript *Script) { return Script->Next; }

sScript *Script_Print(sScript *Script)
{
    MessageBox(NULL, Script->Entries[0].Text, "Text", MB_OK);
    return Script->Next;
}

```

Как видите, все волшебство сосредоточено в рекурсивной функции **Script\_IfThen**, обрабатывающей действия скрипта, находящиеся между парой **if...then** и действием **EndIf**. Действие **Else** просто переключает режим обработки (с пропуска действий на их выполнение или наоборот) в зависимости от исходного значения переменной **Skipped**.

Теперь вы почувствовали мощь, а если вам нужны еще доказательства, обратитесь к последующим главам, где используется система MLS, например, к главе 12, «Управление игроками и персонажами», и главе 16,



«Объединяем все в законченную игру». Обе главы демонстрируют использование скриптов при взаимодействии с игровыми персонажами.

## Применение скриптов в играх

Начиная проект приготовьтесь реализовать поддержку скриптов почти во всех частях игры. Например, скрипты могут пригодиться когда вы работаете с диалогами и видеовставками, и вплоть до эффектов заклинаний и работы с имуществом игрока. Фактически, создание игрового движка, позволяющего использовать скрипты практически для всех игровых данных, приводит к открытому и эффективному проекту.

В главе 16 вы узнаете как применять скрипты в различных компонентах вашей игры, таких, как сражения и инвентарная система. А пока, если вы хотите лучше познакомиться со всеми концепциями скриптов, исследуйте программу **MlsDemo**, находящуюся на прилагаемом к книге CD-ROM.

## Заканчиваем со скриптами

Представленная в этой главе система скриптов при правильном использовании очень мощна, и ее будет достаточно для большинства ваших игровых проектов. Продвинутые читатели, желающие разработать собственный «настоящий» скриптовый язык (например, похожий на C++), могут приобрести хорошую книгу по компиляторам, особенно ту, где используются **lex** и **yacc** (две программы, обрабатывающие текст и грамматику). Одна из таких книг, с названием «**lex & yacc**» является замечательным руководством для изучения основ создания процессоров разбора скриптовых языков. За дополнительной информацией о книге обратитесь к приложению А, «Список литературы».

Если вы заинтригованы мощностью системы **MLS**, можете перед началом проекта создать набор шаблонов действий, которые будут использоваться на протяжении всей игры. В этой главе я обсудил только простые техники для этого, но уверен, что основываясь на предоставленной информации вы придумаете другие замечательные применения для **MLS**.

### Программы на CD-ROM

Программы, демонстрирующие обсуждавшийся в этой главе код, находятся на прилагаемом к книге CD-ROM. В папке **\BookCode\Chap10\** вы найдете следующие программы:

**MlsEdit** — редактор скриптов **Mad Lib Script**, замечательно подходящий для разработки скриптов к вашему проекту.  
Местоположение: **\BookCode\Chap10\MlsEdit\**.

**MlsDemo** — небольшой проект, демонстрирующий выполнение скриптов **Mad Lib**, созданных в редакторе **MLS Editor**.  
Местоположение: **\BookCode\Chap10\MlsDemo\**.

# Глава 11

## Определение и использование объектов

Большие вещи, маленькие вещи, круглые вещи и множество других вещей — мир заполнен объектами различных размеров, форм и назначения. Я говорю не только о нашем мире — мир вашей игры также должен содержать полезные предметы. Отслеживание всех этих полезных объектов и их назначения в вашей ролевой игре является сложной работой, но вооружившись знаниями вы сможете без помех приняться за задание!

В главе вы узнаете следующее:

- Определение объектов в вашей игре.
- Создание главного списка предметов.
- Использование инвентарной системы для управления предметами.

### Определение объектов для вашей игры

Я отчаянно роюсь в своем мешке. Я клал вчера в него целебный эликсир; куда он подевался? Нашел время терять — в середине битвы, получая удары со всех сторон, и вот, уличив момент, я не могу поправить здоровье!

Посмотрим, вот мой кинжал, дополнительная защита, горстка золотых монет, что-то непонятное и — ох, вот он — мой целебный эликсир! И как я смог собрать все это барахло? Ладно, разберусь позже; сейчас я должен глотнуть эликсира и вернуться к убийству монстров.

К счастью, это испытание не угрожало моей жизни; достаточно было на минуту приостановить игру, отсортировать список моего имущества и найти необходимый эликсир. Восстановив здоровье и возобновив игру, я продолжил сражение как истинный воин!

В ходе игры вы собираете различные предметы (называемые также *объекты*), у каждого из которых есть уникальное назначение. Создавая игру вы должны перечислить все объекты, указав для чего каждый из них используется. Оружие, броня, лекарства — все должно быть описано. Для этого нужна форма и функция.

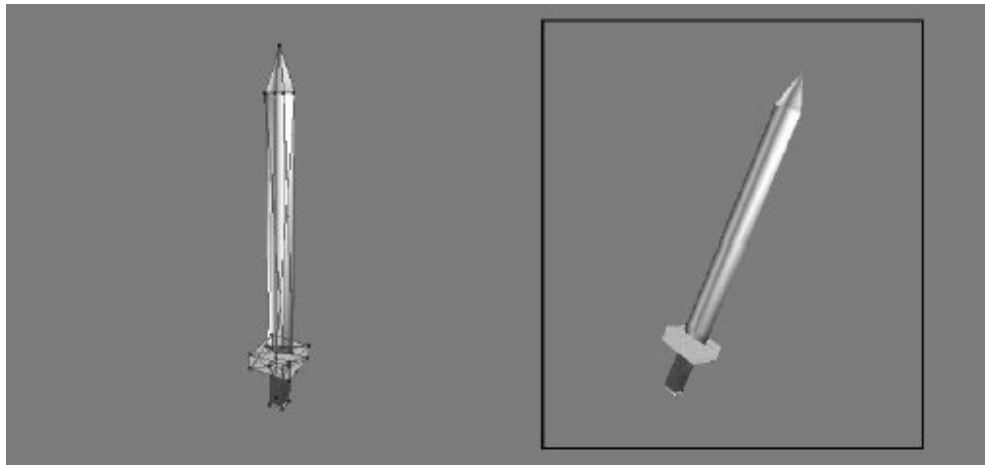
Форма и функция — два главных слова в определении объектов. *Форма* ссылается на представление и идентификацию — как выглядит объект, на что он похож, насколько большой, сколько весит и т.д. *Функция* описывает назначение; у каждого объекта есть назначение — монеты для покупки, меч для сражений, эликсиры для лечения.

В этом разделе вы узнаете, как описать форму и функцию объектов в формате, удобном для использования в вашем игровом проекте.

## Форма объекта

Хотя для нас форма является основополагающей для идентификации объекта, для компьютера она ничего не значит, и достаточно, чтобы объект был представлен в виде изображения или трехмерной модели. В главе 2, «Рисование с DirectX Graphics», вы узнали как просто загрузить растровое изображение или X-файл, содержащий трехмерную сетку, так почему бы не использовать эти растры или сетки для описания формы объекта?

Предположим, вы хотите создать оружие, или, более конкретно, меч. В трехмерной игре (например, той, которую я описывал в начале главы), вы захотите, чтобы игрок видел, что его персонаж несет меч и мог исследовать этот меч, взглянув на него поближе. Кроме того, чтобы показать экипировку (что несет персонаж), вам надо отображать на экране растровое изображение меча. Для представления меча вам потребуется одна сетка и одно растровое изображение (рис. 11.1).



*Рис. 11.1. Показанную слева сетку вы можете использовать в игровом экране (с держащим ее персонажем) и в списке имущества (для исследования меча). Растровое изображение справа отображается во время игры, чтобы показать, что игрок держит меч*

Знаю, что вы подумали, — какой большой и тяжелый меч на рис. 11.1! В настоящем мире меч «большой и тяжелый». Впрочем, вы же не хотите, чтобы никто не смог поднять его. Меч может весить около трех килограммов и быть чуть больше метра длинной. Если в вашей игре учитываются физические свойства объектов, каждому предмету в игре должны быть назначены размеры и вес. Я предпочитаю измерять размер в кубических дециметрах.

**ПРИМЕЧАНИЕ**

Зачем в игре беспокоиться о весе и размере предметов? Вам нужен в игре подросток, размахивающий огромным мечом? А как насчет крошечного эльфа, пытающегося нести двухметровый меч и поклажу, которая в два раза больше чем он сам? Многие игры игнорируют эти проблемы, но если вы хотите, чтобы ваша игра выглядела реалистично, учитывайте вес и размер.

Теперь можно начать встраивать информацию о предметах в игру. Начнем с создания структуры, которая будет хранить информацию о мече (да и о любом другом предмете):

```
typedef struct sItem
{
    char Name[32];           // Краткое название предмета
    char Description[128];   // Описание предмета

    float Weight;            // Вес (в кг.)
    float Size;              // Размер (в куб. дециметрах)

    char MeshFilename[16];   // Имя X-файла с сеткой
    char ImageFilename[16];  // Имя файла .BMP
} sItem;
```

**ПРИМЕЧАНИЕ**

Обратите внимание на 16-байтное ограничение размера буферов **MeshFilename** и **ImageFilename** — это значит, что вы можете использовать имена файлов длиной только 16 символов (15 плюс завершающий символ **NULL**). Если вы считаете, что вам потребуются более длинные имена, просто увеличьте размер буферов до требуемого.

Описание объекта **Sword** в игре будет выглядеть так:

```
sItem Sword = {
    "Меч", "Большой тяжелый меч.",
    2.25f, 12.68f,
    "Sword.x", "Sword.bmp"
};
```

Теперь меч готов к использованию! Ладно, не по-настоящему, поскольку сейчас мы задали только физические свойства меча (вместе с именами файлов для сетки и для изображения). Движку игры недостаточно этой информации чтобы использовать предмет. Здесь на сцену выходит функция предмета.

## Определение функции объекта

Если какой-нибудь предмет попадет в руки ребенка (да и любого из нас), ребенок найдет ему применение, хотя, возможно, оно и не будет правильным. Может показаться заманчивым сделать объектами в игре все типы предметов, но это не соответствует вашей цели. Предметы (объекты) в вашей игре будут использоваться по своему назначению, которое может

быть отнесено к одной из следующих категорий (помните, что это не полный список, а лишь отправная точка для ваших идей):

- **Аксессуары.** Кольца, ожерелья, пояса и любые другие виды носимого снаряжения, которые могут помочь персонажу, классифицируются как аксессуары. Иногда аксессуар — это магическая часть снаряжения, увеличивающая способности того, кто его носит.
- **Броня.** Вы не будете терять здоровье от ударов, если носите соответствующую защиту. Броня может быть самой разной — от полных доспехов до пары ботинок.
- **Коллекции.** Любые предметы, у которых нет специального назначения, считаются коллекционными. Они могут играть важную роль в сценарии игры, но могут быть и абсолютно бесполезными. Картины, декоративные статуэтки, кресла — все это коллекционные предметы.
- **Еда.** Запеченные устрицы, пироги с устрицами, устрицы по-французски, устрицы в сливочном соусе и даже устрицы барбекю — все, что пожелаете. Персонажи вашей игры должны периодически испытывать голод, и все, что они могут найти и съесть, классифицируется как еда, в том числе микстуры и травы.
- **Деньги.** Все, что относится к сфере финансов, независимо от того, какую форму оно принимает — монеты, банкноты, раковины и т.д. Играм не требуется деноминация — одна золотая монета так же хороша, как и другая, и чем больше у вас монет, тем лучше.
- **Транспорт.** Автобусы, лодки, самолеты, дельтапланы: люди знают как перемещаться по миру и предметы для этого принадлежат персонажам игры. Лодку нельзя носить с собой, но ваша игра распознает персонажа, являющегося владельцем предмета, делая соответствующую запись в его инвентаре.
- **Оружие.** Меч, камень или удавка — любая вещь, которую можно использовать для нанесения повреждений.
- **Прочее.** Все, что не попадает в перечисленные выше категории, классифицируется как «прочее».

Ваш игровой движок определяет, что делает каждый из предметов, на основании категории предмета. Например, оружие можно добавить к экипировке (вооружиться им), еду можно употребить в пищу. Чтобы сделать работу с предметами легче, можно при необходимости добавить подкатегории. Например, лечебный эликсир, хотя и съедобный, может быть отнесен к категории лекарственных предметов. Когда ваш игровой движок видит лекарственный предмет, он сразу знает, что надо увеличить уровень здоровья персонажа.

У каждого предмета есть дополнительная информация для использования, так что мы не можем остановиться здесь. У оружия есть

атрибут силы, увеличивающий способность персонажа наносить повреждения, лечебный эликсир восстанавливает здоровье, а броня уменьшает повреждения, получаемые персонажем в сражении. Теперь давайте взглянем, что из себя представляет каждая категория предметов.

## Оружие

Персонажи могут наносить повреждения голыми руками. Чем сильнее персонаж, тем больше повреждений он может нанести. Дайте персонажу в руки оружие и количество наносимых им повреждений резко возрастет. Одни виды оружия более смертоносны, чем другие, так что поиски лучшего оружия это одна из весомых причин для того, чтобы участвовать в приключениях.

У оружия может быть несколько применений. Например, зачарованный меч может ударить скрывающегося демона и, в то же самое время, создать мощную шаровую молнию. Такие дополнительные функции предметов я называю *специализациями (specials)*.

Персонаж не может использовать любое оружие; здесь есть ряд ограничений. Размеры, вес, мощность и цена — вот некоторые из ограничений, которые надо учитывать. Какое удовольствие, если в руках новичка окажется самое мощное оружие в игре? Поэтому вы добавляете ограничения использования (больше об этом вы узнаете в разделе «Ограничения использования» далее в этой главе).

Вернемся назад к тому факту, что одни виды оружия могут наносить больше повреждений, чем другие. Количество наносимых оружием повреждений определяется числом, называемым *модификатор атаки (attack modifier)*. Чем больше число, тем больший ущерб наносит оружие. Кроме того, некоторые виды оружия проще в обращении, что позволяет вам чаще поражать цели. Чтобы определить, насколько легко поразить цель данным оружием, вы используете *модификатор поражений (to-hit modifier)*. Чем больше модификатор поражений оружия, тем больше шансов, что персонаж, используя данное оружие, поразит цель. Об использовании модификаторов и о том, как эти модификаторы относятся к персонажам, вы подробнее узнаете в главе 12, «Управление игроками и персонажами».

### ПРИМЕЧАНИЕ

*Модификатор (modifier)* — это нечто, меняющее каким-либо образом атрибуты персонажа. Например, *модификатор повреждений (damage modifier)* уменьшает или увеличивает количество повреждений.

Некоторые типы оружия могут быть объединены в специальные группы, называемые *группы вооружений (weapons groups)*. Отдельные виды оружия могут наносить конкретным созданиям больший вред, чем другие — для примера подумайте об использовании огненного меча против ледяного монстра.

И, наконец, вооружение может быть разделено на такие группы, как оружие ближнего боя (*hand-to-hand*) и оружие дальнего боя (*ranged*).

Независимо от типа, у каждого оружия есть *радиус действия* (*range of use*). Меч может поразить цель, находящуюся прямо перед вами, в то время, как стрела лука может попасть в цель, расположенную в десятках метров от вас. Кроме того, некоторые виды оружия могут поражать сразу несколько целей. Все эти вещи следует учесть при разработке оружия.

## **Броня**

Чем больше защиты, тем лучше, и в игре помогает каждый кусочек брони. Броня помогает повысить сопротивляемость к повреждениям. Количество сопротивляемости называется *модификатором защиты* (*defense modifier*). Также как и у вооружения, у брони есть специальные возможности, ограничения использования и возможность объединения в *группы защиты* (*armor groups*).

Броня может быть разделена на множество подкатегорий, таких как шлемы, нагрудники, кольчуги, поножи, ботинки, перчатки и т.д.

## **Аксессуары**

Как упоминалось ранее, у аксессуаров обычно есть специфические области применения. Магическое кольцо можно надеть для того, чтобы стать невидимым. Вы можете стать невидимым сразу, как только наденете кольцо, или, возможно, невидимость будет требовать активации. Аксессуары могут также действовать как броня; они могут увеличивать сопротивляемость некоторым воздействиям в игре — например, повышать сопротивляемость к ядам.

## **Еда**

Съедобные предметы обычно бывают нескольких видов. Пища поддерживает жизнь, лекарства увеличивают здоровье, яды уменьшают здоровье. Снова могут оказывать эффект специальные возможности, но, поскольку у съедобных предметов всего несколько применений, вы можете жестко запрограммировать их в движке игры.

## **Коллекции**

Коллекционные предметы обычно просты; они нужны только для какого-нибудь маленького фрагмента игры. Например, если персонаж дает вам свой портрет, чтобы вы отнесли его девушке из соседнего города (и только для этой цели), портрет считается коллекционным предметом. Коллекционные предметы просто способствуют развитию сюжета игры. Возможно, отправивший портрет персонаж получит в свою очередь какой-нибудь специальный предмет от девушки из соседнего города.

## **Транспорт**

Перемещаться пешком медленно и скучно, поэтому могут потребоваться другие виды транспорта. Назначение транспорта — изменять способ

перемещения персонажа (обычно по карте). Транспортные средства могут также открывать новые области в игре, которые были прежде недоступны. Например, приобретенная вашим персонажем лодка может использоваться чтобы переплыть через озеро к уединенному острову, или, возможно, лошадь, которую вы видели в соседнем городе, поможет вам благополучно пересечь бесплодную пустыню.

## Прочее

«Прочие» предметы большей частью бесполезны, поскольку не имеют какого-либо применения. Однако не стоит выбрасывать их. По меньшей мере, они могут выполнять определенные действия, заданные в движке. Например, если ваш персонаж хорошо проявит себя в битве, его могут наградить медалью. Хотя эта медаль и замечательно выглядит, она не служит никакой цели в игре.

## Добавление функций к объекту

В предыдущем разделе, «Определение функций объекта», вы увидели как много надо описать в функции объекта. К счастью, поскольку объект должен быть отнесен к одной из категорий, требуется не вся информация — меч наносит повреждения, а броня защищает — так что не требуется смешивать данные атаки и защиты.

## Категории предметов и значения

Фактически, вам потребуется однозначно категоризировать каждый предмет в соответствии с вашим игровым движком, как это сделал я в разделе «Определение функций объекта». Каждая категория объектов нумеруется для ссылок (1 — оружие, 2 — броня и т.д.). У каждой категории есть связанные с ней значения, они могут определять модификатор (атака или защита), специальное использование, количество наносимых повреждений или добавляемого здоровья и присоединенный скрипт. Верно. Предметы могут использовать скрипты для увеличения своих возможностей.

За исключением присоединенного скрипта, все остальные значения могут быть представлены одной переменной — единой для значения модификатора, добавляемого здоровья и всего остального. Итак, добавим к ранее созданной структуре **sItem** следующие две переменные:

```
// ... Предыдущая информация sItem

long Category; // 1-5 представляет показанные выше
                // категории предметов
long Value;    // Модификатор, увеличение здоровья и т.д.

// ... Дальнейшая информация sItem
```



**СОВЕТ**

Для значения, представляющего категорию предмета в структуре **sItem** можно использовать следующее перечисление:

```
enum ItemCategories {
    WEAPON = 0,
    ARMOR,
    SHIELD,
    HEALING,
    OTHER
};
```

**Задание цены предметов**

У всего в игре есть своя цена. Назначение цены каждому предмету помогает определить, что игрок может купить или продать и по какой цене. Не стоит загромождать каждый предмет множеством цен; просто укажите единственное значение, которое персонаж должен заплатить, чтобы купить данный предмет. При продаже того же самого предмета персонажем ему будет предложена цена несколько меньшая той, которую он должен был бы заплатить, если бы покупал этот предмет. Например, предметы могут продаваться за половину той цены, которую персонаж заплатил при их покупке.

Цена предмета может быть добавлена в структуру **sItem** следующим образом:

```
// ... Предыдущая информация sItem

long Price; // Цена покупки предмета
```

**Флаги предмета**

Иногда, вы можете захотеть, чтобы у игрока не было возможности продавать некоторые предметы — например, важные магические артефакты. Об этом позаботится битовый флаг и сейчас мы сразу добавим еще несколько флагов. В таблице 11.1 перечислены некоторые флаги, которые вы можете использовать.

**Таблица 11.1.** Битовые флаги предметов

Флаг	Описание
<b>SELLABLE</b>	Предмет может быть продан.
<b>CANDROP</b>	Предмет может быть выброшен. Не используйте этот флаг для важных предметов, если не хотите, чтобы персонаж игрока мог их выбросить.
<b>USEONCE</b>	Предмет может быть использован только один раз. После использования предмет пропадает.
<b>UNKNOWN</b>	Неизвестный предмет. Чтобы правильно использовать его вы должны выполнить опознание.

Все флаги хранятся в переменной, расположенной в структуре **sItem**:

```
long Flags; // Битовые флаги предмета

// ... Дальнейшая информация sItem
```

Пример вы можете посмотреть в замечательной игре Phantasy Star Online от Sega. Каждый флаг может быть представлен элементом перечисления (тогда получится максимум 32 флага). Чтобы установить, очистить или проверить флаг, используйте следующие макросы (в макросах **v** представляет переменную, хранящую флаги предмета, а **f** — интересующий вас флаг):

```
enum {
    SELLABLE = 0, // Бит 0
    CANDROP,      // Бит 1
    USEONCE,      // Бит 2
    UNKNOWN       // Бит 3
};

#define SetItemFlag(v,f)  (v |= (1 << f))
#define ClearItemFlag(v,f) (v &= ~(1 << f))
#define CheckItemFlag(v,f) (v & (1 << f))

// Пример использования макросов и флагов
long ItemFlags = 0;

// Устанавливаем флаги, указывающие, что предмет
// может быть продан и брошен
SetItemFlag(ItemFlags, SELLABLE);
SetItemFlag(ItemFlags, CANDROP);

// Проверяем, может ли предмет быть брошен
// и отображаем сообщение
if(CheckItemFlag(ItemFlags, CANDROP))
    MessageBox(NULL, "Can Drop Item", "Item", MB_OK);

// Очищаем флаг продажи
ClearItemFlag(ItemFlags, SELLABLE);
```

## Ограничения использования

Вы можете захотеть, чтобы некоторые персонажи вашей игры не могли использовать отдельные предметы. Например, маг не может пользоваться тяжелым двуручным боевым топором, а для варвара бесполезен магический посох. В таких случаях, когда только конкретный персонаж может пользоваться конкретным предметом, вам необходимы ограничения использования для заданных классов персонажей.

### ПРИМЕЧАНИЕ

*Класс персонажа (character class)* — это классификация или группировка персонажей в зависимости от их расы или профессии. Например, все люди относятся к одному классу, но если быть более точным, человек-воин и человек-маг относятся к разным классам (или просто воин и маг — кто сказал, что они должны быть людьми?).

Чтобы представить ограничения использования элемента, в структуру **sItem** включается еще одна переменная, отслеживающая 32 бита информации. Каждый бит представляет собой отдельный класс, значит всего вы можете отслеживать 32 класса. Если предмет может использоваться каким-либо классом, соответствующий бит установлен, если для персонажей данного класса предмет бесполезен — представляющий класс бит сброшен.

Вот дополнение к структуре **sItem** для поддержки ограничений использования:

```
long Usage; // Ограничения использования

// ... Другие данные sItem
```

Чтобы было проще устанавливать, очищать и проверять биты классов в ограничениях использования, можно применять следующие макросы (**v** представляет переменную с флагами, а **c** — номер класса в диапазоне от 0 до 31):

```
#define SetUsageBit(v,c)    (v |= (1 << c))
#define ClearUsageBit(v,c) (v &= (~(1 << c)))
#define CheckUsageBit(v,c) (v & (1 << c))

// Пример использования макросов
long Flags = 0;

SetUsageBit(Flags, 5);      // Устанавливаем бит класса 5

if(CheckUsageBit(Flags, 5)) // Проверяем бит класса 5
    MessageBox(NULL, "Usage Set", "Bit", MB_OK);

ClearUsageBit(Flags, 5);    // Очищаем бит класса 5
```

Используя показанные макросы (**SetUsageBit**, **ClearUsageBit** и **CheckUsageBit**) вы можете быстро проверить, может ли персонаж использовать предмет экипировки, основываясь на классе этого персонажа. Предположим, в игре маги относятся к классу 1, а воины — к классу 2. Когда маг попытается вооружиться палахом (у которого бит для класса 1 сброшен), игровой движок сообщит игроку, что маг не может использовать этот предмет.

### **Присоединение скриптов к предметам**

Чтобы сделать предметы более универсальными, к ним можно прикреплять скрипты. Скрипт срабатывает при каждом использовании предмета — когда персонаж выпивает эликсир, когда в битве применяется меч или когда пользователь активирует специальную функцию предмета (например, использует волшебную палочку).

#### **СОВЕТ**

При использовании скриптов хорошо применять специальный шаблон действий, приспособленный для предметов. За дополнительной информацией о создании шаблона действий и использовании редактора скриптов обратитесь к главе 10, «Реализация скриптов».

Сейчас вам надо только сохранить в структуре **sItem** имя файла скрипта.

```
// .. Предыдущие данные sItem

char ScriptFilename[16]; // Имя файла скрипта .mls
```

## Сетки и изображения

Скорее всего, вы захотите, чтобы игрок мог видеть, как выглядит предмет, а значит вам потребуется загрузить представляющее объект двухмерное изображение или трехмерную сетку. Для этого внесите следующие дополнения в структуру **sItem**:

```
// .. Предыдущие данные sItem

char MeshFilename[16]; // Имя файла сетки .X
char ImageFilename[16]; // Имя файла изображения .bmp
} sItem; // Завершение структуры
```

## Итоговая структура данных предмета

Теперь структура **sItem** готова к использованию! Взгляните на нее целиком (включая вспомогательные макросы):

```
enum ItemCategories { WEAPON = 0,
                      ARMOR,
                      SHIELD,
                      HEALING,
                      OTHER
                    };

#define SetUsageBit(v,c)  (v |= (1 << c))
#define ClearUsageBit(v,c) (v &= (~(1 << c)))
#define CheckUsageBit(v,c) (v & (1 << c))

enum {
    SELLABLE = 0, // Бит 0
    CANDROP,     // Бит 1
    USEONCE,     // Бит 2
    UNKNOWN      // Бит 3
};

#define SetItemFlag(v,f)  (v |= (1 << f))
#define ClearItemFlag(v,f) (v &= ~(1 << f))
#define CheckItemFlag(v,f) (v & (1 << f))

typedef struct sItem
{
    char Name[32];           // Краткое название предмета
    char Description[128];   // Описание предмета

    float Weight;            // Вес (в кг.)
    float Size;              // Размер (в куб. дм.)

    long Category;           // Категория предмета
    long Value;              // Модификатор, увеличение здоровья и т.д.
    long Price;              // Цена покупки предмета
}
```

```
long Flags;           // Битовые флаги предмета
long Usage;           // Ограничения использования

char ScriptFilename[16]; // Имя файла скрипта .mls
char MeshFilename[16];   // Имя файла сетки .X
char ImageFilename[16];  // Имя файла изображения .bmp
} sItem;
```

---

**ПРИМЕЧАНИЕ** Помните, что размер буферов **ScriptFilename**, **MeshFilename** и **ImageFilename** ограничен 16 байтами, а значит, в структуре **sItem** вы можете хранить только имя файла без пути к нему. Кроме того, хотя структура **sItem** и хранит имена файлов, за их загрузку для использования в игре отвечаете вы.

Если вы хотите хранить пути к файлам или использовать длинные имена файлов для скриптов, сеток или изображений, измените соответствующим образом размер буферов.

---

Получив законченную структуру **sItem** можно вернуться к построению структуры данных меча. Предположим, что у меча модификатор повреждений равен +10 (это означает, что в битве к коэффициенту повреждений будет добавляться 10). Меч в игре обычно стоит 200 монет и может использоваться только воинами (класс 2).

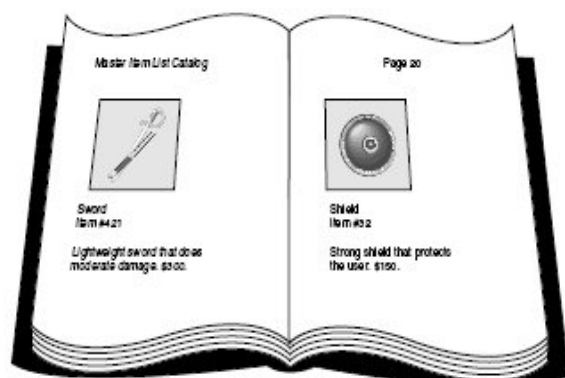
```
// Определение классов персонажей
#define WIZARD 1
#define WARRIOR 2

sItem Sword = {
    "Меч", "Большой тяжелый меч", // Название и описание
    2.25f, 12.68f, // Вес и размер
    WEAPON, 200, SELLABLE | CANDROP, // Категория, цена и флаги
    (1 << WARRIOR), // Используется классом 2 (воинами)
    "", "Sword.x", "Sword.bmp" // Файлы скрипта, сетки и изображения
};
```

Теперь меч описан и вы можете использовать его в игре. Но что за польза от единственного предмета? Мир вашей игры должен быть заполнен объектами! Как же обращаться со всеми ними?

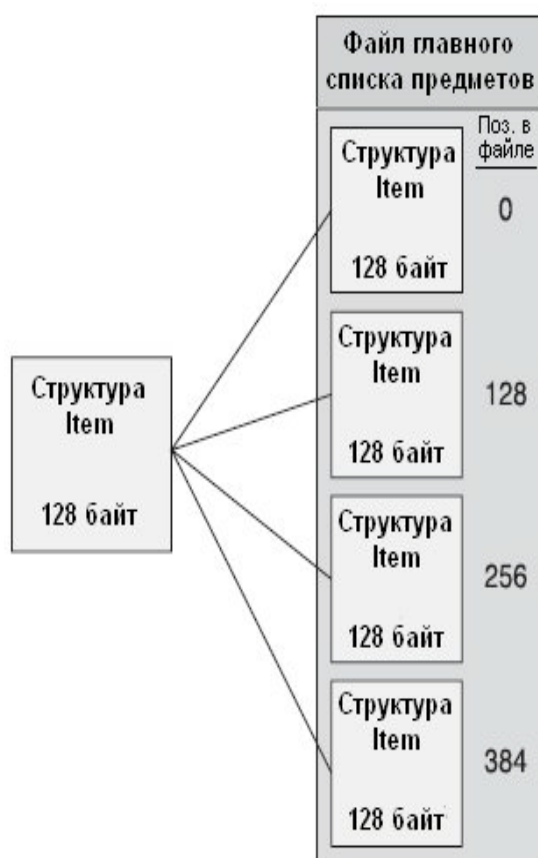
## Главный список предметов

Каждый предмет в игре должен быть описан, и чтобы не запутаться в них, надо держать все описания в *главном списке предметов (master item list, MIL)*. Думайте о MIL, как о каталоге товаров, подобном изображенному на рис. 11.2. У каждого объекта есть номер для ссылок, и хранится только одно описание предмета (даже если в игре таких предметов несколько).



*Рис. 11.2. Подобно каталогу товаров, главный список предметов помогает держать объекты вашего мира в порядке*

В любое время, когда вам требуется новый объект, или вы хотите получить атрибуты какого-либо объекта, вы обращаетесь к МЛ. На самом нижнем уровне МЛ вашей игры может храниться как массив структур **sItem** или единый последовательный файл, состоящий из списка структур элементов (подобно тому, что показано на рис. 11.3). Как же создать свой собственный МЛ? Давайте посмотрим.



*Рис. 11.3. Последовательный файл МЛ отслеживает каждый предмет в вашей игре. Размер данных каждого предмета фиксирован, что обеспечивает быстрый доступ к файлу в случае необходимости получить данные*

## Конструирование МЛ

Приведенный ниже фрагмент кода создает небольшую структуру данных предмета, содержащую название предмета, его вес и размер. Это та же самая структура **sItem**, с которой начиналась эта глава, — мы просто повторно определили ее для хранения информации в главном списке предметов. Мы используем эту структуру для создания простого МЛ:

```
typedef struct sItem
{
    char   Name[32]; // Название предмета
    float  Weight;   // Вес (в кг.)
    float  Size;     // Размер (в куб. дм.)
};
```

Теперь предположим, что вы хотите сохранить в MIL пять предметов, представив их в виде массива структур **sItem**:

```
sItem Items[5] = {
    { "Большой меч", 2.25f, 12.68f },
    { "Малый меч", 0.9f, 6.34f },
    { "Волшебная палочка", 0.22f, 3.17f },
    { "Камень", 0.45f, 1.57f },
    { "Эликсир", 0.22f, 1.57f }
};
```

Определив ваш собственный MIL (используя массив структур **sItem**), вы захотите сохранить список в файле для последующего использования. Этот случай имеет место, когда вы используете отдельную программу, создающую для вас MIL, такую, как вы увидите в следующем разделе «Использование редактора MIL». Сейчас взгляните на фрагмент кода, который создает файл (с названием *items.mil*) и сохраняет в него массив **Items**:

```
FILE *fp=fopen("items.mil", "wb");

for(short i = 0; i < 5; i++)
    fwrite(&Items[i], 1, sizeof(sItem), fp);

fclose(fp);
```

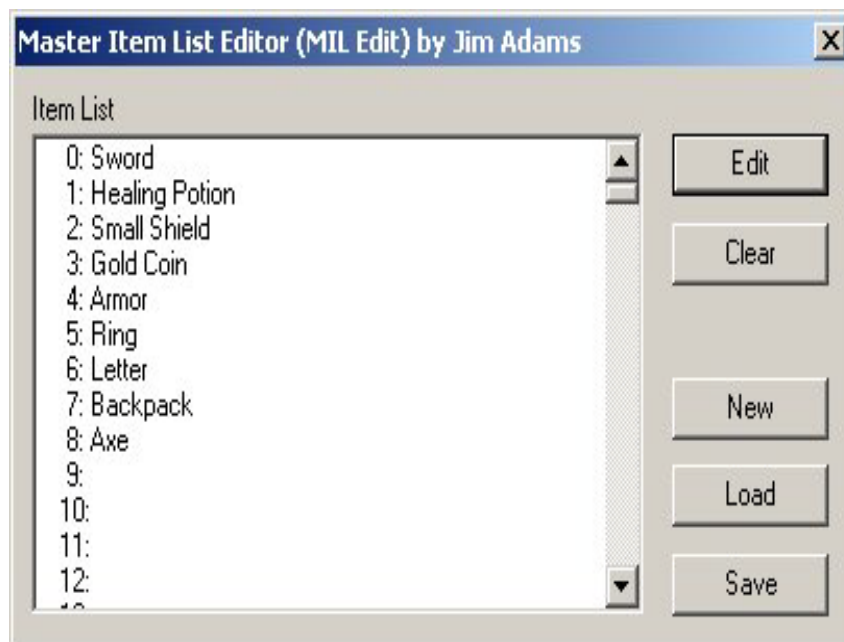
Хотя показанный пример создания файла MIL короткий и ясный, он абсолютно не годится для реальных приложений, в частности для ролевых игр. Описание предмета должно содержать больше информации, и вы, возможно, будете работать с тысячами предметов. Делать все это вручную в коде — напрасная трата времени. Вам нужен редактор предметов, который поможет в создании и управлении MIL — итак, встречайте *MIL Editor*.

## Использование редактора MIL

Необходимость быстрого и простого создания предметов привела к появлению редактора главного списка предметов (редактора MIL). Во многом похожий на обсуждавшийся в главе 10, «Реализация скриптов», редактор MLS Editor, MIL Editor также состоит из единственного главного окна, позволяющего перемещаться по списку предметов и редактировать атрибуты каждого предмета. Вы можете сохранять и загружать MIL, но список атрибутов элементов останется фиксированным (пока вы не перепрограммируете редактор для соответствия вашим потребностям).

Полный исходный код редактора MIL Editor содержится на прилагаемом к книге CD-ROM (в каталоге \BookCode\Chap11\MILEdit). Когда вы запустите находящийся на CD-ROM редактор MIL Editor, появится

диалоговое окно **Master Item List Editor**, показанное на рис. 11.4. В этом диалоговом окне есть окно со списком, содержащим все предметы, и кнопки для редактирования данных предмета и для сохранения и загрузки списков предметов. В списке предметов предусмотрено место для 1024 объектов, а это значит, что номер предмета можно хранить в 16-разрядной переменной (в диапазоне от 0 до 1023).

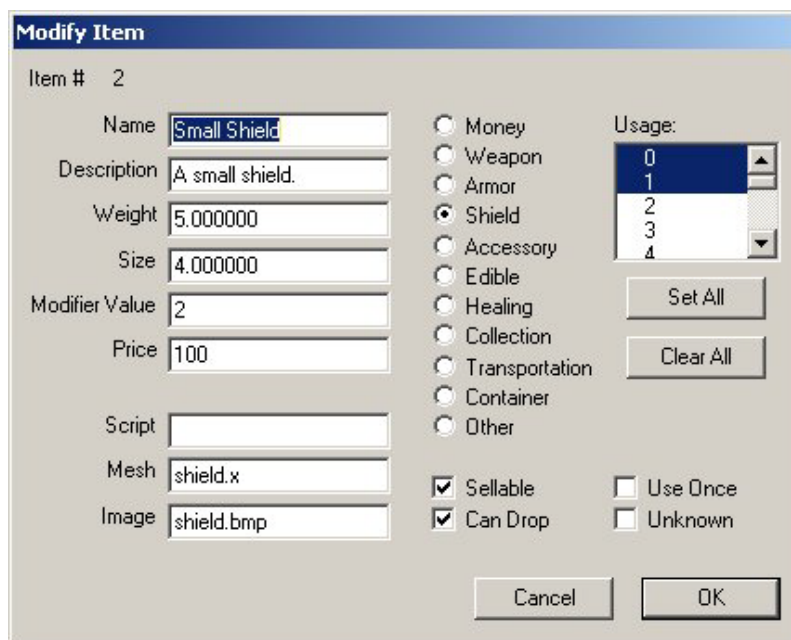


*Рис. 11.4. Редактор главного списка предметов используется для создания и редактирования данных игровых объектов. В показанном списке определено восемь предметов*

Чтобы начать работу с редактором главного списка предметов найдите и запустите файл **MILEdit.exe** (он находится в папке **\BookCode\Chap11\MILEdit**). В диалоговом окне **Master Item List Editor** вы можете выполнять следующие действия для добавления или редактирования предметов и записи их списка на диск:

1. Выберите предмет в списке **Item List** дважды щелкнув по нему (либо отметив предмет и щелкнув по кнопке **Edit**), или добавьте новый предмет, щелкнув по кнопке **Add**. На экран будет выведено диалоговое окно **Modify Item** (рис. 11.5).
2. Отредактируйте требуемые поля в диалоговом окне **Modify Item**.
3. Завершив редактирование щелкните **OK**, чтобы применить изменения и вернуться к диалоговому окну **Master Item List Editor**.
4. Чтобы сохранить результаты работы, щелкните по кнопке **Save** в диалоговом окне **Master Item List Editor**, и в диалоговом окне **Save MIL File** введите имя файла и запишите MIL на диск.





**Рис. 11.5.** Диалоговое окно *Modify Item* позволяет вам редактировать данные каждого объекта

#### ПРИМЕЧАНИЕ

Загрузка MIL также проста, как и запись; щелкните по кнопке **Load** в диалоговом окне **Master Item List Editor**. Затем в появившемся диалоговом окне **Load MIL File** выберите файл, который хотите загрузить.

Файлы MIL обычно используют расширение .MIL. Вы можете для своих списков использовать другое расширение, но примеры программ из этой книги используют .MIL.

Редактор MIL Editor использует ту же самую версию структуры **sItem**, что была показана ранее в этой главе, но я добавил дополнительные категории предметов. Эти дополнительные категории — *Защита (Shield)*, *Лечение (Healing)* и *Контейнер (Container)*, объекты, в которых можно переносить другие предметы, например, рюкзак). Эти дополнительные категории добавлены в список перечисления **ItemCategories**, показанный ранее в разделе «Категории предметов и значения»:

```
enum ItemCategories
{
    MONEY = 0, WEAPON,    ARMOR,
    SHIELD,    ACCESSORY, EDIBLE,
    HEALING,   COLLECTION, TRANSPORTATION,
    CONTAINER, OTHER
};
```

Если вы решите модифицировать редактор MIL Editor для использования других атрибутов предметов или категорий, не забудьте соответствующим образом изменить структуру **sItem**. Теперь вы можете начать использовать созданные вами данные предметов в вашем игровом проекте.

## Доступ к предметам из MIL

Если у вас есть MIL, можно загрузить весь список целиком в массив структур **sItem**, используя следующий код:

```
sItem Items[1024]; // Массив структур sItem

// Открываем файл Default.mil
FILE *fp = fopen("Default.mil", "rb");

// Читаем данные всех предметов
for(short i = 0; i < 1024; i++)
    fread(&Items[i], 1, sizeof(sItem), fp);

// Закрываем файл
fclose(fp);
```

Здесь я предполагаю, что ваша структура данных предмета относительно мала, и в вашем списке MIL не больше 1024 предметов. Что произойдет, если вы расширите структуру данных **sItem** или решите хранить в MIL больше предметов? Это приведет к значительному расходу памяти.

Чтобы предотвратить загрузку данных каждого предмета из MIL в память, вы можете обращаться к данным отдельных предметов непосредственно в файле MIL. Поскольку размер каждой структуры данных предмета фиксирован, вы можете получить доступ к данным отдельного предмета, перемещая указатель доступа к файлу на соответствующую позицию и считывая структуру, как показано в следующем фрагменте кода:

```
// ItemNum = номер загружаемого предмета
sItem Item;

// Открываем файл MIL с именем items.mil
FILE *fp=fopen("items.mil", "rb");

// Перемещаемся на соответствующую позицию в файле,
// основываясь на размере структуры sItem и номере
// загружаемого предмета
fseek(fp, sizeof(sItem) * ItemNum, SEEK_SET);

// Читаем структуру данных предмета
fread(&Item, 1, sizeof(sItem), fp);

// Закрываем файл
fclose(fp);
```

Вот он — быстрый и простой доступ к каждому предмету в MIL! Теперь пришло время что-нибудь сделать со всеми этими предметами.

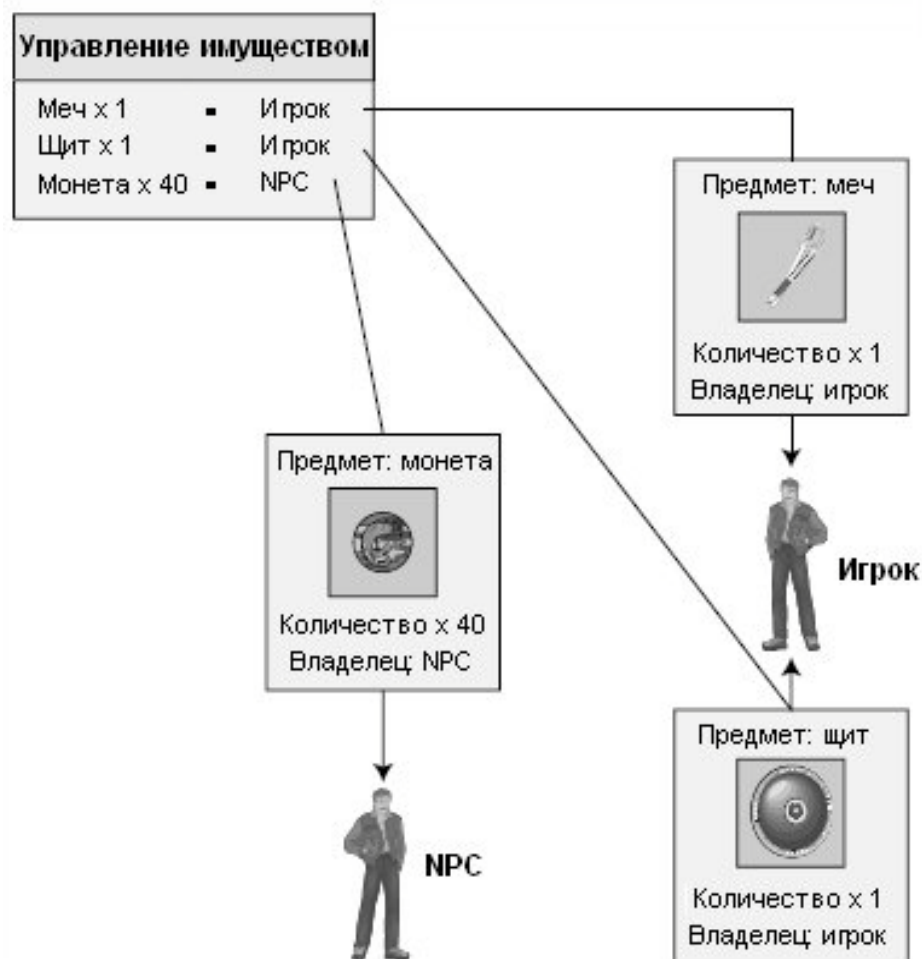
## Управление предметами с помощью системы контроля имущества

Ваши предметы готовы быть разбросанными по миру, и только вопрос времени, как скоро игроки начнут пытаться утащить все, что не прибито

гвоздями. Когда это произойдет, игрокам потребуется способ управления предметами, что включает применение следящей за предметами *системы контроля имущества (Inventory Control System, ICS)*.

Не заблуждайтесь; ICS применяется не только к персонажам игроков, она следит за всем миром. Предметы могут лежать на карте, принадлежать персонажам или даже находиться внутри других предметов (например, предметы могут быть сложены в рюкзак). Это означает, что предмету должен быть назначено *право собственности (ownership)*. Кроме того, у владельца может быть *несколько экземпляров* объекта — например, несколько монет.

Собрание предметов владельца называется *списком имущества (inventory list)*, и любой объект может находиться в этом списке (а также несколько экземпляров одного объекта). Взаимоотношения между владельцами, списками имущества и количеством предметов показаны на рис. 11.6. (*Примечание: не думайте о предметах, как об относящихся к владельцу; вместо этого думайте о владельце, как об имеющем набор предметов.*)



**Рис. 11.6.** Система управления имуществом следит за каждым владельцем предметов, в том числе и за количеством предметов

Система управления имуществом работает рука об руку с MIL. В то время как MIL содержит только единственное, уникальное описание каждого

объекта в мире, ICS может работать с несколькими экземплярами одного и того же объекта. Каждый раз, когда системе управления имуществом требуется информация о предмете, она обращается к главному списку предметов за его спецификациями. Это позволяет экономить память, сохраняя в ICS только ссылки с номерами объектов в MIL (как показано на рис. 11.7).



*Рис. 11.7. Если система управления имуществом хранит только номера для ссылок на предметы, можно сэкономить значительный объем памяти, разместив всю информацию о предмете в главном списке предметов*

Для игровых карт и уровней замечательно подойдет простая система управления имуществом (называемая *ICS карты*, *map ICS*), хранящая только список предметов и их местоположение на карте, поскольку она позволит разместить повсюду предметы, чтобы игроки могли собирать их. Настоящая проблема возникнет, когда игроки начнут собирать разбросанные предметы, добавляя их к своему имуществу. Накапливаются дублирующие экземпляры, добавляются новые предметы, а другие предметы используются или выбрасываются. Предметы быстро приходят в беспорядок. Обработка наборов находящихся у персонажей объектов является работой системы управления имуществом персонажей, которая несколько сложнее, чем вариант, относящийся к карте.

## Разработка ICS карты

ICS карты следит за предметами, находящимися внутри уровней, включая те предметы, которые находятся внутри других, — например, меч лежащий в сундуке. Способ размещения предметов на карте зависит от используемого типа карты. На трехмерных картах для размещения предметов вы используете три координаты — X, Y и Z. Если вы разделили ваш мир на несколько карт, придется следить за предметами каждой карты, используя собственный файл предметов для каждой карты.

ICS карты представлена структурой и классом:

```
typedef struct sMapItem
{
    long ItemNum;           // Номер предмета в MIL
    long Quantity;         // Количество предметов
                           // (например, монет)
    float XPos, YPos, ZPos; // Координаты на карте

    sMapItem *Prev, *Next; // Указатели связанного списка

    long Index;             // Индекс данного предмета
    long Owner;             // Индекс владельца
    sMapItem *Parent;       // Родитель предмета

    sMapItem()
    {
        Prev = Next = Parent = NULL;
        Index = 0; Owner = -1;
    }

    ~sMapItem() { delete Next; Next = NULL; }
} sMapItem;

class cMapICS
{
private:
    long m_NumItems;        // Количество предметов на карте
    sMapItem *m_ItemParent; // Родитель связанного списка
                           // предметов карты

    // Чтение из файла длинного целого числа
    // или числа с плавающей точкой
    long GetNextLong(FILE *fp);
    float GetNextFloat(FILE *fp);

public:
    cMapICS(); // Конструктор
    ~cMapICS(); // Деструктор

    // Загрузка, сохранение и освобождение
    // списка предметов карты
    BOOL Load(char *Filename);
    BOOL Save(char *Filename);
    BOOL Free();

    // Добавление и удаление предмета на карте
    BOOL Add(long ItemNum, long Quantity,
             float XPos, float YPos, float ZPos,
             sMapItem *OwnerItem = NULL);
    BOOL Remove(sMapItem *Item);

    // Возвращает количество предметов или
    // родителя связанного списка объектов
    long GetNumItems();
    sMapItem *GetParentItem();
    sMapItem *GetItem(long Num);
};
```

Сначала вы видите структуру **sMapItem**, которая хранит информацию для каждого предмета на карте. **ItemNum** — это номер для ссылки на элемент MIL (который должен находиться в диапазоне от 0 до 1023, если вы

используете программу MIL Editor), а **Quantity** — это количество предметов **ItemNum** (это позволяет, например, представить горсть монет как единый объект). Затем расположены координаты предмета на карте **XPos**, **YPos** и **ZPos**.

Далее следуют указатели **Prev** и **Next**. Они добавлены для формирования связанного списка структур **sMapItem**. Следующая пара переменных, **Index** и **Owner**, используются при загрузке и сохранении предметов на карте. **Index** хранит текущий индекс предмета в связанном списке. Если предмет принадлежит другому предмету, переменная **Owner** хранит индекс родительского объекта (в противном случае значение **Owner** равно  $-1$ ).

При загрузке (или добавлении) объекта вы устанавливаете заключительную переменную в **sMapItem (Parent)**, чтобы она указывала на структуру данных действительного владельца предмета. Концепция связанного списка структур **sMapItem** показана на рис. 11.8.



*Рис. 11.8. Вы храните список предметов карты в виде связанного списка. В процессе чтения с диска предметы нумеруются, что помогает устанавливать отношения родитель-потомок между объектами*

Структура **sMapItem** использует и конструктор и деструктор, вызываемые когда для структуры выделяется память и при освобождении занятых ресурсов соответственно. Обе функции обеспечивают правильность указателей связанного списка и, когда структура удаляется, все последующие структуры **sMapItem** в связанном списке удаляются тоже.

**ВНИМАНИЕ!**

Если вы хотите удалить из связанного списка только один экземпляр структуры **sMapItem**, сперва присвойте переменной **next** удаляемого экземпляра значение **NULL**. Это гарантирует, что все последующие экземпляры в связанном списке не будут удалены.

В классе **cMapICS** есть две закрытые функции (**GetNextLong** и **GetNextFloat**) используемые для чтения текста и преобразования его в значение типа **long** или **float**. Также в классе **cMapICS** есть восемь полезных открытых функций. Давайте познакомимся с ними поближе.

### ***cMapICS::Load, cMapICS::Save и cMapICS::Free***

Согласно своим именам, это трио функций загружает, сохраняет и освобождает список относящихся к карте предметов. Первая из трех, **Load**, создает и загружает список предметов. Для простоты, все предметы хранятся в текстовом файле, использующем следующий формат:

```
MIL_ItemNum
Quantity
XPos
YPos
ZPos
ParentID
```

Каждый предмет использует шесть строк текста, и каждый элемент (группа из шести строк) последовательно нумеруется (первый предмет в файле — это предмет 0, второй предмет — предмет 1 и т.д.). Вот пример файла, содержащего два предмета:

```
// Описание предмета 0:
10      // Номер предмета в MIL (значение long)
1       // Количество (значение long)
10.0    // XPos (значение float)
0.0     // YPos (значение float)
600.0   // ZPos (значение float)
-1      // Владелец (-1 = нет, иначе его индекс)
// Описание предмета 1:
1       // Номер предмета в MIL
1       // ...
10.0
0.0
600.0
0       // Относится к предмету 0 (первому предмету в файле)
```

Комментарии добавлены только для ясности; в реальном файле их не будет. Читая список предметов, такой как показан выше, функция **Load** преобразует текст в числа. Из этих чисел создается структура **sMapItem** для каждого загруженного предмета на карте и формируется связанный список загруженных предметов. После считывания данных каждого предмета выполняется сопоставление связанных между собою предметов (с использованием указателя **Parent** в структуре **sMapItem**).

Здесь нет ничего по-настоящему сложного, так что давайте перейдем прямо к коду **cMapICS::Load**:

```

BOOL cMapICS::Load(char *Filename)
{
    FILE *fp;
    long LongNum;

    sMapItem *Item, *ItemPtr = NULL;

    Free(); // Освобождаем предыдущий набор

    // Открываем файл
    if((fp=fopen(Filename, "rb"))==NULL)
        return FALSE;

```

Сразу после открытия файла (показанного в предыдущем фрагменте кода), вы можете начинать загружать данные предметов. Первый фрагмент данных, который вам надо загрузить — это номер предмета. Как только номер предмета загружен, вы создаете новую структуру данных предмета карты и присоединяете ее к связанному списку предметов. Кроме того, одни предметы могут быть связаны с другими (например, эликсир может находиться в котомке, и в этом случае эликсир и котомка связаны между собою). Процесс продолжается, пока не будут загружены все предметы.

```

// Бесконечный цикл чтения данных предметов
while(1) {
    // Получаем номер следующего предмета
    // (прерываемся, если больше нет предметов,
    // о чем свидетельствует возвращаемое значение -1)
    if((LongNum = GetNextLong(fp)) == -1)
        break;

    // Создаем новый указатель карты и привязываем его
    Item = new sMapItem();
    if(ItemPtr == NULL)
        m_ItemParent = Item;
    else {
        Item->Prev = ItemPtr;
        ItemPtr->Next = Item;
    }
    ItemPtr = Item;

    // Сохраняем номер предмета в MIL
    Item->ItemNum = LongNum;

    // Получаем количество
    Item->Quantity = GetNextLong(fp);

    // Получаем координаты
    Item->XPos = GetNextFloat(fp);
    Item->YPos = GetNextFloat(fp);
    Item->ZPos = GetNextFloat(fp);

    // Получаем номер владельца
    Item->Owner = GetNextLong(fp);

    // Сохраняем индекс и увеличиваем счетчик
    Item->Index = m_NumItems++;
}

```



```
    }

    // Закрываем файл
    fclose(fp);

    // Сопоставляем взаимосвязанные объекты
    ItemPtr = m_ItemParent;
    while(ItemPtr != NULL) {
        // Проверяем, относится ли предмет к другому
        if(ItemPtr->Owner != -1) {
            // Находим соответствующий родительский предмет
            Item = m_ItemParent;
            while(Item != NULL) {
                if(ItemPtr->Owner == Item->Index) {
                    // Связываем, устанавливая указатель на родителя
                    ItemPtr->Parent = Item;
                    break; // Останавливаем сканирование родителей
                }
                Item = Item->Next;
            }
        }
        // Переходим к следующему предмету
        ItemPtr = ItemPtr->Next;
    }
    return TRUE;
}
```

---

**ПРИМЕЧАНИЕ** Подобно большей части кода из этой книги, функции класса **cMapICS** возвращают **TRUE**, если функция завершена успешно, и **FALSE**, если произошла ошибка.

---

Функция **Save** берет внутренний список предметов и, используя указанное вами имя файла, записывает этот список в файл на диске. Функция **Save** обычно используется для обновления игровых данных, поскольку игроки могут подбирать одни предметы и выбрасывать другие.

Сперва функция **Save** назначает каждой структуре **sMapItem** в связанном списке значение индекса (основываясь на их порядке следования). Первому элементу связанного списка присваивается индекс 0, второму — индекс 1 и т.д. Затем обновляются переменные **Owner** во всех дочерних объектах и данные записываются в файл:

```
BOOL cMapICS::Save(char *Filename)
{
    FILE *fp;
    sMapItem *Item;
    long Index = 0;

    // Открываем файл
    if((fp=fopen(Filename, "wb")) == NULL)
        return FALSE;

    // Назначаем индексы предметам
    if((Item = m_ItemParent) == NULL) {
        fclose(fp);
        return TRUE; // Нет предметов для сохранения
    }
}
```

```

while(Item != NULL) {
    Item->Index = Index++;
    Item = Item->Next;
}

// Сопоставляем дочерние предметы и родителей
Item = m_ItemParent;
while(Item != NULL) {
    if(Item->Parent != NULL)
        Item->Owner = Item->Parent->Index;
    else
        Item->Owner = -1;
    Item = Item->Next;
}

// Сохраняем все
Item = m_ItemParent;

while(Item != NULL) {

    // Номер предмета
    fprintf(fp, "%lu\r\n", Item->ItemNum);

    // Количество
    fprintf(fp, "%lu\r\n", Item->Quantity);

    // Координаты
    fprintf(fp, "%lf\r\n%lf\r\n%lf\r\n",
        Item->XPos, Item->YPos, Item->ZPos);

    // Номер владельца
    fprintf(fp, "%ld\r\n", Item->Owner);

    // Следующий предмет
    Item = Item->Next;
}
fclose(fp); // Закрываем файл
return TRUE; // Возвращаем флаг успеха!
}

```

И, наконец, функцию **Free** вы используете, когда уничтожаете класс (удаляя таким образом связанный список предметов). Вот код для **Free**:

```

BOOL cMapICS::Free()
{
    m_NumItems = 0;
    delete m_ParentItem;
    m_ParentItem = NULL;
    return TRUE;
}

```

Вы просто удаляете связанный список предметов и подготавливаете класс для нового использования.

### ***cMapICS::Add и cMapICS::Remove***

Когда на карте появляется новый предмет (например, брошенный игроком), вам необходимо вызвать функцию **Add**, чтобы этот брошенный предмет был включен в связанный список объектов карты. Функция **Add** начинается с

создания новой структуры **sMapItem**, затем она заполняет эту структуру предоставленной вами информацией о предмете и включает структуру в список объектов карты:

```
BOOL cMapICS::Add(long ItemNum, long Quantity,
                  float XPos, float YPos, float ZPos,
                  sMapItem *OwnerItem)
{
    sMapItem *Item;

    // Создание новой структуры данных предмета
    Item = new sMapItem();

    // Вставляем ее в вершину списка
    Item->Next = m_ItemParent;
    if(m_ItemParent != NULL)
        m_ItemParent->Prev = Item;
    m_ItemParent = Item;

    // Заполняем структуру данных
    Item->ItemNum = ItemNum;
    Item->Quantity = Quantity;
    Item->XPos = XPos;
    Item->YPos = YPos;
    Item->ZPos = ZPos;
    Item->Parent = OwnerItem;

    return TRUE;
}
```

Функцию **Add** вы вызываете чтобы добавить предмет к списку объектов карты, и точно так же вы используете функцию **Remove** для удаления предметов с карты. Вызывая **Remove** вы указываете идентификатор предмета, который хотите удалить из списка карты. Помимо этого **Remove** освобождает память, занятую структурой данных предмета, и заботится о тех предметах, которые были связаны с удаляемым:

```
BOOL cMapICS::Remove(sMapItem *Item)
{
    sMapItem *ItemPtr, *NextItem;

    // Сперва удаляем дочерние предметы
    if((ItemPtr = m_ItemParent) != NULL) {
        while(ItemPtr != NULL) {
            NextItem = ItemPtr->Next;
            if(ItemPtr->Parent == Item)
                Remove(ItemPtr);
            ItemPtr = NextItem;
        }
    }

    // Удаляем из связанного списка и сбрасываем
    // корень, если это текущая голова списка
    if(Item->Prev != NULL)
        Item->Prev->Next = Item->Next;
    else
        m_ItemParent = Item->Next;
}
```

```

    if(Item->Next != NULL)
        Item->Next->Prev = Item->Prev;

    // Очищаем связанный список
    Item->Prev = Item->Next = NULL;

    // Освобождаем память
    delete Item;

    return TRUE;
}

```

### ***cMapICS::GetNumItems, cMapICS::GetParentItem и cMapICS::GetItem***

Вы используете эти три функции для получения количества относящихся к карте предметов, получения родительской структуры **sMapItem** и получения указанной структуры данных предмета из связанного списка. Первые две функции возвращают единственную переменную, а третья функция выполняет сложную работу по сканированию связанного списка объектов и возвращает указанный элемент списка:

```

long cMapICS::GetNumItems() { return m_NumItems; }

sMapItem cMapICS::GetParentItem() { return m_ParentItem; }

sMapItem *cMapICS::GetItem(long Num)
{
    sMapItem *Item;

    // Начинаем с родительского предмета
    Item = m_ItemParent;

    // Цикл, пока не достигнут предмет с указанным номером
    while(Num-) {
        if(Item == NULL)
            return NULL; // Нет больше предметов для сканирования,
                        // возвращаем ошибку
        Item = Item->Next; // Переходим к следующему предмету
                        // в связанном списке
    }
    return Item; // Возвращаем найденный предмет
}

```

#### **ПРИМЕЧАНИЕ**

С указателем на родительскую структуру, возвращаемым **GetParentItem**, вы можете просканировать весь связанный список предметов, используя указатель **Next** каждой структуры. Если вам надо получить заданную структуру данных предмета, основываясь на ее индексе в списке, используйте функцию **GetItem**.

### ***Использование класса cMapICS***

У каждой карты в вашей игре есть связанный с ней список принадлежащих этой карте предметов. ICS карты загружает эти предметы и предоставляет их движку, когда надо визуализировать карту или добавить какой-нибудь

предмет к имуществу игрока (если он подбирает находящийся на карте предмет).

Взгляните на фрагмент кода, загружающий пример списка предметов, добавляющий один предмет, удаляющий другой предмет и сохраняющий список:

```
cMapICS MapICS;

MapICS.Load("sample.mi"); // Загрузка файла

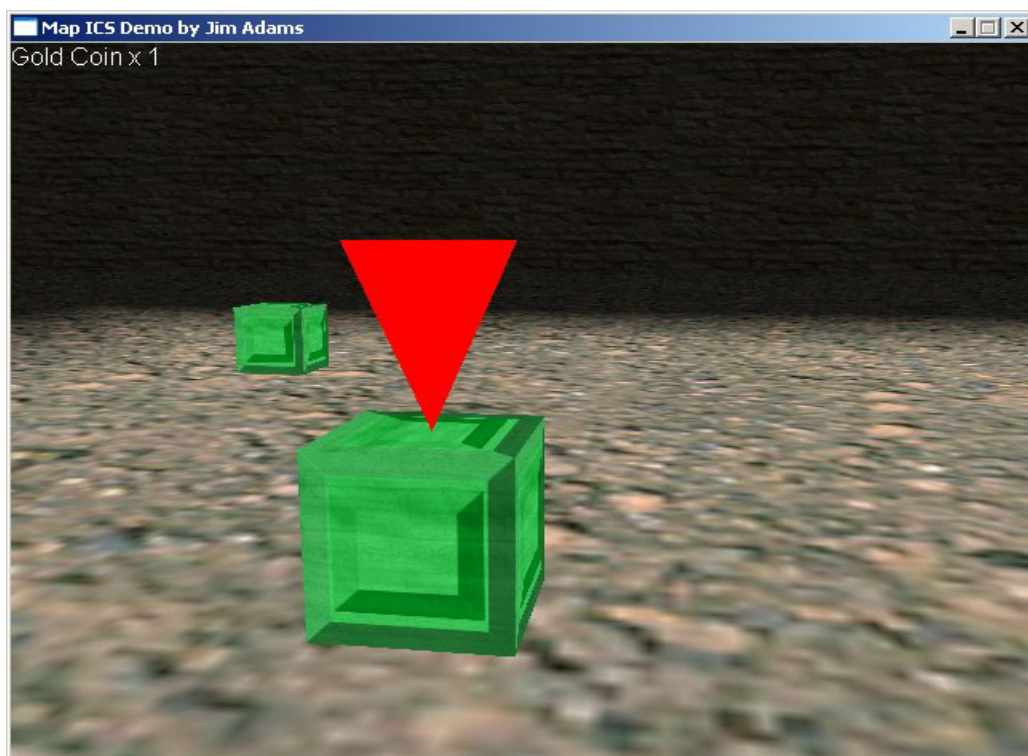
// Добавляем один предмет с номером 10
MapICS.Add(10, 1, 0.0f, 0.0f, 100.0f, NULL);

// Удаляем второй предмет в списке
MapICS.Remove(MapICS.GetItem(1));

// Сохраняем список
MapICS.Save("sample.mi");
```

Это очень простой пример модификации списка предметов карты, но что мешает нам двинуться дальше и посмотреть, насколько сложным он может быть?

Демонстрационная программа MapICS (рис. 11.9) содержит полный код класса **cMapICS**, показанный в этой главе и структуру **sItem** из программы MIL Editor. Вы можете загрузить программу MapICS с прилагаемого к книге CD-ROM (она находится в папке \BookCode\Chap11\MapICS). В ней используется ICS карты и MIL для визуализации набора объектов, разбросанных по небольшому уровню.



*Рис. 11.9. Программа MapICS позволяет вам перемещаться по небольшому уровню, подбирать и бросать предметы*

Программа MapICS загружает объекты карты и использует данные об их координатах для рисования сеток предметов в сцене. Всякий раз, когда вы приближаетесь к предмету, появляется значок цели и отображается название предмета.

В программе MapICS нет ничего нового для вас, поэтому я не буду приводить здесь ее код. Фактически, MapICS очень похожа на программу NodeTree, которую мы рассматривали в главе 8, «Создание трехмерного графического движка».

---

<b>ПРИМЕЧАНИЕ</b>	Программа MapICS позволяет вам перемещаться по простому уровню, используя клавиши управления курсором и мышь. Вы можете подобрать предмет, встав перед ним и нажав левую кнопку мыши. Если вы нажмете правую кнопку мыши, будет выброшен случайно выбранный предмет.
-------------------	--

---

После визуализации уровня сканируются все предметы карты и рисуются те из них, которые находятся в поле зрения. Затем определяется ближайший предмет и на экран выводится его название. Код хорошо прокомментирован, так что у вас не должно возникнуть проблем при его исследовании.

## Разработка ICS персонажа

Хотя разработка системы управления имуществом персонажа сначала может вызвать испуг, позвольте мне заверить вас, что она не слишком отличается от разработки системы управления имуществом карты. Вам нужна возможность добавлять и удалять предметы, но вам не приходится иметь дело с координатами предметов на карте. Вместо этого ICS игрока следит за порядком, то есть позволяет игроку переупорядочивать предметы, как ему удобно.

Конечно, упорядочивание это просто вариант сортировки связанного списка, но как быть с теми предметами, которые могут содержать внутри себя другие, например, котомки? Если вы в программе MIL Editor указали, что эти предметы являются контейнерами, то беспокоиться не о чем.

Что касается категорий, настоящее волшебство начинается, когда вы хотите использовать предмет или применить его для экипировки. Поскольку у каждого предмета есть категория, ICS персонажа может быстро определить, что делать с рассматриваемым предметом. Если предмет — оружие, ICS персонажа может спросить, надо ли добавить его к экипировке. Если это лекарство, игрок может сразу принять его. Начали улавливать идею?

И, наконец, ICS персонажа позволяет игроку исследовать объекты, для чего предназначены параметры файла сетки и файла изображения в MIL Editor. Когда игрок исследует объект, загружается и отображается соответствующая ему сетка или изображение.

Теперь переключим внимание на объединение всех описанных частей.

## **Определение класса *sCharICS***

Определение разработанных для этой книги класса ICS персонажа и вспомогательных структур выглядит так:

```
typedef struct sCharItem
{
    long ItemNum;           // Номер предмета в MIL
    long Quantity;         // Количество предметов
                           // (например, для монет)
    sCharItem *Prev, *Next; // Указатели связанного списка

    long Index;            // Индекс данного предмета
    long Owner;            // Индекс владельца

    sCharItem *Parent;      // Родитель предмета

    sCharItem()
    {
        Prev = Next = Parent = NULL;
        Index = 0; Owner = -1;
    }

    ~sCharItem() { delete Next; Next = NULL; }
} sCharItem;

class cCharICS
{
private:
    long m_NumItems;        // Количество предметов в имуществе
    sCharItem *m_ItemParent; // Родитель связанного списка предметов

    // Функции для чтения чисел из файла
    long GetNextLong(FILE *fp);
    float GetNextFloat(FILE *fp);

public:
    cCharICS(); // Конструктор
    ~cCharICS(); // Деструктор

    // Загрузка, сохранение и освобождение списка предметов
    BOOL Load(char *Filename);
    BOOL Save(char *Filename);
    BOOL Free();

    // Добавление и удаление предмета
    BOOL Add(long ItemNum, long Quantity,
             sCharItem *OwnerItem = NULL);
    BOOL Remove(sCharItem *Item);

    // Получение количества предметов или
    // родителя связанного списка объектов
    long GetNumItems();
    sCharItem *GetParentItem();
    sCharItem *GetItem(long Num);

    // Функции переупорядочивания
    BOOL Arrange();
    BOOL MoveUp(sCharItem *Item);
    BOOL MoveDown(sCharItem *Item);
};
```

Подобно классу **cMapICS**, класс **cCharICS** использует специальную структуру (**cCharItem**), которая хранит номер предмета в MIP, количество предметов и управляет связанным списком. В отличие от структуры **sMapItem**, **sCharItem** не заботится о координатах предмета.

Точно так же и класс **cCharICS** во многом похож на **cMapICS**, за исключением трех дополнительных открытых функций — **Arrange**, **MoveUp** и **MoveDown**. Вы используете эти функции для сортировки списка предметов персонажа. Вот их код:

```

BOOL cCharICS::Arrange()
{
    sCharItem *Item, *PrevItem;

    // Начинаем с вершины связанного списка и каждый
    // предмет всплывает вверх, пока не наткнется на
    // предмет с меньшим ItemNum. Прерываем работу,
    // достигнув низа списка.
    Item = m_ItemParent;

    while(Item != NULL) {
        // Проверяем наличие предыдущего предмета
        if(Item->Prev != NULL) {
            // Всплываем, пока у предыдущего предмета
            // меньше значение ItemNum или пока не достигнем
            // вершины списка
            while(Item->Prev != NULL) {
                // Получаем указатель на предыдущий предмет
                PrevItem = Item->Prev;

                // Прерываемся, если некуда всплывать
                if(Item->ItemNum >= PrevItem->ItemNum)
                    break;

                // Обмен
                if((PrevItem = Item->Prev) != NULL) {
                    if(PrevItem->Prev != NULL)
                        PrevItem->Prev->Next = Item;
                    if((PrevItem->Next = Item->Next) != NULL)
                        Item->Next->Prev = PrevItem;
                    if((Item->Prev = PrevItem->Prev) == NULL)
                        m_ItemParent = Item;
                    PrevItem->Prev = Item;
                    Item->Next = PrevItem;
                }
            }
            // Переход к следующему объекту
            Item = Item->Next;
        }
        return TRUE;
    }
}

BOOL cCharICS::MoveUp(sCharItem *Item)
{
    sCharItem *PrevItem;

    // Обмениваем предмет с предыдущим
    if((PrevItem = Item->Prev) != NULL) {

```



```

        if(PrevItem->Prev != NULL)
            PrevItem->Prev->Next = Item;
        if((PrevItem->Next = Item->Next) != NULL)
            Item->Next->Prev = PrevItem;
        if((Item->Prev = PrevItem->Prev) == NULL)
            m_ItemParent = Item;
        PrevItem->Prev = Item;
        Item->Next = PrevItem;
    }
    return TRUE; // Возвращаем флаг успеха
}

BOOL cCharICS::MoveDown(sCharItem *Item)
{
    sCharItem *NextItem;

    // Обмениваем элемент с последующим
    if((NextItem = Item->Next) != NULL) {
        if((Item->Next = NextItem->Next) != NULL)
            NextItem->Next->Prev = Item;
        if((NextItem->Prev = Item->Prev) != NULL)
            Item->Prev->Next = NextItem;
        else
            m_ItemParent = NextItem;
        NextItem->Next = Item;
        Item->Prev = NextItem;
    }
    return TRUE; // Возвращаем флаг успеха
}

```

Функция **Arrange** сортирует связанный список предметов по номеру каждого предмета в MIL, от меньших к большим. Если же вам необходимо собственное особое упорядочивание списка, воспользуйтесь функциями **MoveUp** и **MoveDown**, которые получают указатель на уже содержащуюся в списке структуру **sCharItem**.

Функция **MoveUp** перемещает указанную структуру **sItem** в связанном списке вверх, а **MoveDown** перемещает указанную структуру в связанном списке вниз. На рис. 11.10 показано использование функций **Arrange**, **MoveUp** и **MoveDown** в простом связанном списке предметов.

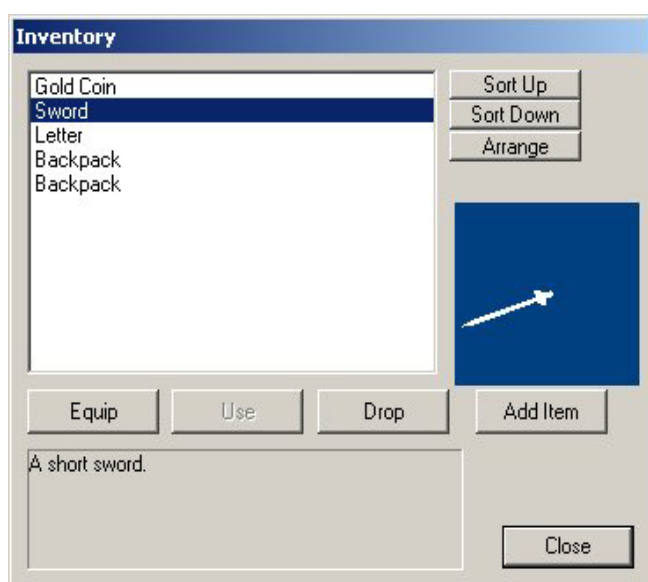


**Рис. 11.10.** Слева направо показаны: исходный неотсортированный список, упорядоченный (отсортированный) список, список после перемещения предмета вверх и список после перемещения предмета вниз

Оставшиеся функции **cCharICS** идентичны функциональности класса **cMapICS**, за очевидным исключением координат предметов, используемых при добавлении предметов в список. Даже формат хранения данных предметов персонажа идентичен формату хранения данных предметов карты, опять же за исключением координат.

### Использование класса **cCharICS**

Чтобы продемонстрировать систему ICS персонажа я создал демонстрационное приложение с именем **CharICS**, которое управляет списком предметов, содержащимся в файле главного списка предметов **default.mil**. Программа находится на прилагаемом к книге CD-ROM (посмотрите в папке **\BookCode\Chap11\CharICS**). При запуске программы отображается диалоговое окно **Inventory** (рис. 11.11). В этом окне показан список предметов воображаемого персонажа, которым вы можете управлять с помощью кнопок.



*Рис. 11.11. Программа CharICS позволяет вам просматривать имущество воображаемого персонажа, добавляя, удаляя или сортируя предметы по своему желанию*

Использование **CharICS** выглядит так:

1. Щелкните по кнопке **Add Item**. Программа **CharICS** добавит случайный предмет из **MIL** в список имущества персонажа.
2. Выберите предмет в списке. Предметы классифицированы, так что кнопки **Use** и **Equip** будут активными или неактивными в зависимости от того, какой предмет вы выбрали. Щелчок по кнопке **Equip** не оказывает никакого эффекта; она вступит в игру позже, когда вы будете иметь дело с реальным персонажем. Щелчок по кнопке **Use** удаляет предмет из списка, если для него установлен флаг **USEONCE**.
3. Щелкните по кнопке **Drop** чтобы выбросить предмет (удалить предмет из списка имущества), если для него установлен флаг **CANDROP**.

4. Чтобы отсортировать предметы в списке имущества, вы можете щелкнуть по предмету, а затем щелкнуть по кнопке **Sort Up** или **Sort Down**. Выбранный предмет переместится в списке вверх или вниз, соответственно. Чтобы отсортировать список предметов по их номерам, щелкните по кнопке **Arrange**.

И, наконец, дополнительное удовольствие, — предметам соответствуют трехмерные сетки, которые отображаются в поле, находящемся в правой части окна приложения. Показываемые трехмерные объекты медленно вращаются, так что вы можете рассмотреть их со всех сторон. Как здорово!

## Заканчиваем с объектами и имуществом

Не позволяйте кажущейся сложности работы с объектами в вашей игре одурачить вас. Факт в том, что эти предметы в вашей игре являются небольшими и легко управляемыми фрагментами данных. Когда персонаж начинает использовать эти предметы, все становится несколько сложнее. В этой главе вы увидели только как использовать системы управления имуществом для контроля за предметами, относящимися к карте и персонажу.

Чтобы увеличить полезность редактора MIL Editor (в том числе и для предметов в вашей собственной игре), я рекомендую вам добавить свои атрибуты предметов. Это потребует знаний программирования для Windows, поскольку придется модифицировать диалоговое окно, чтобы оно соответствовало внесенным изменениям. Кроме того, надо будет переписать определение структуры **sItem**, чтобы в ней хранились добавленные вами новые атрибуты. Конечно, не стоит беспокоиться об этом вызове; вы становитесь мастером программирования ролевых игр.

### Программы на CD-ROM

Программы, демонстрирующие обсуждавшийся в этой главе код, находятся на прилагаемом к книге CD-ROM. В папке \BookCode\Chap11\ вы найдете следующие программы:

**MILEdit** — обсуждавшаяся в главе программа редактора главного списка предметов. Редактор помогает вам создать предметы, которые вы будете использовать в своем игровом проекте. Местоположение: \BookCode\Chap11\MILEdit\.

**MapICS** — пример системы управления имуществом карты. Предметы загружаются с диска и разбрасываются по карте, где пользователь может перемещаться, используя мышь и клавиши управления курсором. Местоположение: \BookCode\Chap11\MapICS\.

**CharICS** — пример системы управления имуществом персонажа, позволяющий просматривать список предметов. Предметы могут исследоваться, использоваться или применяться для экипировки. Местоположение: \BookCode\Chap11\CharICS\.

# Глава 12

## Управление игроками и персонажами

Игровой мир будет ничем без бродящих по нему игроков и монстров. Однако создание их сперва может пугать. Не волнуйтесь. В этой главе вы найдете основную информацию, о том как создать персонажи и дать им собственную жизнь в вашей игре — все в простом для следования виде.

В главе вы научитесь делать следующее:

- Определение персонажей.
- Управление персонажами в игре.
- Общение с персонажами.
- Использование скриптов для персонажей.
- Использование персонажами заклинаний.
- Взаимодействие между персонажем игрока и независимыми персонажами.
- Обработка сражений.

### Игроки, персонажи, монстры — о боже!

До этой главы мы рассматривали весьма стандартные игровые компоненты. Все эти компоненты — графический движок, система управления объектами и скрипты — очень легко поддерживать. Теперь ситуация может показаться тупиковой, поскольку формирование персонажей может быть как легким, так и трудным, в зависимости от вашей игры.

Вы должны работать с персонажем игрока (который для краткости будет называться PC, player character), находящимся под управлением игрока, независимыми персонажами (их называют NPC, non-player character), которые ходят вокруг, населяя мир, и ведут собственную жизнь, и монстрами (называемыми MC, monster character), которые только и хотят убить вас.

В этом разделе я познакомлю вас с различными темами, которые помогут вам проектировать и определять игровых персонажей, включая управление ими, их умения и особенности.

**ПРИМЕЧАНИЕ**

*Персонаж игрока (player character)* — это альтер-эго игрока. Независимые персонажи — это персонажи, над которыми игрок не имеет непосредственного контроля. Часто NPC — это горожане, торговцы, банкиры, лекари и монстры. Да, даже монстры являются NPC, но в этой книге я провожу четкое различие между NPC и MC. NPC не атакуют игрока, а MC — атакуют.

---

Поговорим о затруднениях! Как обрабатывать каждый крошечный аспект этих типов персонажей? Вы можете сначала разбить эти аспекты на следующие категории, а затем определить, как обрабатывать каждую из них:

- **Определение.** Используя набор правил, вы определяете все, что данный персонаж может совершать в игре. Эти правила должны точно определять, что персонаж в состоянии сделать. Определение персонажа включает его способности (такие, как сила и проворность) и графическое представление персонажа (например, в виде трехмерной сетки).
- **Перемещение.** Персонажи должны быть способны перемещаться по миру, пешком, на лошади или на самолете.
- **Общение.** Персонаж игрока взаимодействует с персонажами других игроков и с независимыми персонажами, разговаривая с ними или общаясь другим способом. Ход вашей игры может измениться от одного произнесенного слова, а история может постоянно развиваться по шаблонам разговоров.
- **Управление ресурсами.** Управление ресурсами очень важно в ролевых играх. У вас есть имущество персонажа, в которое входят доспехи, оружие, эликсиры, ключи и все прочее, что персонаж может положить в свои виртуальные карманы. Эти предметы, конечно, надо использовать в какой-то из моментов игры, и задача персонажа — определить, как применять каждый из объектов. Заклинания и магия также попадают в эту категорию.
- **Сражения.** Сражения — это средство избавления мира от злодеев, которые являются NPC или другими PC, дающее вашему персонажу шанс увеличить свои опыт и силу. Помните, что главный элемент ролевой игры — взять здорового, крутого персонажа и сделать его еще здоровее и круче!
- **Развитие персонажа.** Как я только что упомянул, цель типичной ролевой игры состоит в непрерывном увеличении силы и опыта вашего персонажа, чтобы он был готов к более сложным уровням вашей игры. По мере увеличения опыта персонаж получает новые умения, силы, заклинания и возможности.
- **Действия персонажа.** Персонажи игры могут многое. Они могут ходить, атаковать, произносить заклинания, использовать предметы и многое другое. Каждому действию, которое может выполнять

персонаж, должно соответствовать графическое представление, например, в виде отображаемой на экране трехмерной сетки.

## Определение персонажей в вашей игре

Развертывание персонажей в вашей игре — одна из наиболее трудных работ, с которой вы встретитесь. Даже на базовом уровне код, необходимый для управления игровыми персонажами, может быть весьма замысловатым. Что же делать скромному программисту? Не торопясь начать работу с самого начала, и все будет хорошо.

В этом разделе я познакомлю вас с несколькими базовыми концепциями, общими для большинства ролевых игр. Эти концепции включают определение базовых способностей персонажа (включая его силу, ум, проворство и т.д.), как персонаж перемещается по игровому миру и как персонажи могут взаимодействовать между собой. Давайте для начала посмотрим, как можно определить способности персонажа.

### *Способности персонажа*

Персонажам нужны различные способности, влияющие на результаты происходящих в игре действий, например, сражений. Если персонаж выхватывает меч, каков шанс, что он попадет в намеченную цель и, если это произошло, какой ущерб нанесет меч?

Типичная ролевая игра присваивает способностям числовые значения — чем больше значение, тем лучше персонаж в терминах конкретной способности. Использование чисел также упрощает вычисления. Например, типичная способность в ролевой игре — сила. Скажем, сила измеряется от 0 (хилак) до 999 (супергерой). Тогда у среднего человека сила должна находиться в диапазоне от 100 до 150.

Сделаем следующий шаг и предположим, что персонаж должен обладать определенной силой, чтобы выполнить какое-то действие. Например, выбивание заклинившей двери требует, чтобы сила человека была 100 или больше. Подъем огромного булыжника требует силы 500. Улавливаете идею?

Какими еще способностями должен обладать персонаж? Это важный для вашей игры вопрос, так что тщательно обдумайте его. Что будет происходить в вашей игре требующее назначения способностей персонажу? В общем случае можно начать со способностей, показанных в таблице 12.1.

Каждой способности в вашей игре назначается числовое значение, и каждая способность по-разному использует это число. Атака — это количество повреждений, наносимых невооруженным персонажем. Если ее значение 100, персонаж атакуя может нанести до 100 очков повреждений. Здесь позвольте мне сказать, что персонаж может получить достаточно много очков повреждений, прежде чем погибнет.

**Таблица 12.1.** Способности персонажа

<b>Способность</b>	<b>Описание</b>
Атака	Количество повреждений, которое может наносить персонаж. Значение атаки базируется на том, сколько повреждений может нанести персонаж голыми руками, но добавление оружия увеличивает значение атаки (в зависимости от типа оружия).
Защита	В полном комплекте доспехов вы чувствуете себя неуязвимым, но когда на вас только джинсы и футболка вы беззащитны. Защита определяет уровень вашей уязвимости — чем больше значение, тем меньше наносимый вам ущерб. У каждого персонажа есть природная защита, будь это кожа или панцирь. Использование брони или других элементов экипировки увеличивает защиту.
Проворность	Способность быстро и ловко перемещаться. Проворный персонаж быстрее передвигается и даже может легко уклониться от атаки.
Интеллект	Способность персонажа контролировать свой разум. Полный контроль над разумом необходим для произносящих заклинания магов и подобных персонажей. Чем выше интеллект, тем больше шансов, что произнесенное персонажем заклинание поразит намеренную цель.
Сопrotивляемость	Защита помогает уменьшить физический вред от физических атак, а сопротивляемость уменьшает вред от магических атак. Чем выше сопротивляемость, тем меньший ущерб наносят вам заклинания. Стопроцентная сопротивляемость делает персонаж невосприимчивым к магии!
Меткость	Некоторые персонажи хорошо целятся, у других это вызывает затруднения. Меткость персонажа определяет насколько успешно он поражает намеренную цель во время атаки. Эта способность увеличивается или уменьшается в зависимости от типа используемого оружия или другой экипировки.
<b>СОВЕТ</b>	Чтобы игра была интереснее можно использовать некий уровень повреждений, когда слабый персонаж атакует хорошо защищенного, не имея возможности причинить ему ущерб. Позже, в разделе «Сражения и персонажи», вы увидите, как модифицировать способность атаковать у слабого персонажа, чтобы он мог нанести какие-то повреждения персонажу с высокой способностью защищаться.

Защита, с другой стороны, сокращает количество повреждений, получаемых при атаке. Если у персонажа значение защиты 50, количество

получаемых при атаке повреждений уменьшается на 50 очков, так что чем лучше защита, тем меньше ущерб.

Вы можете изменять способности атаки и защиты с помощью предметов. Отдельные предметы, такие как оружие, имеют собственные значения, которые добавляются к значению атаки персонажа (снова, чтобы больше узнать об этих модификаторах, обратитесь к разделу «Сражения и персонажи»).

В терминах проворности, у персонажа есть врожденный шанс уклониться от атаки, который увеличивается по мере того, как персонаж становится сильнее. Проворность измеряется от 0 до 999 — чем больше значение, тем выше шанс уклониться от атаки. Чтобы определить шансы уклонения от атаки берется случайное число и сравнивается со значением проворности. Если случайное число меньше или равно значению способности проворства, персонаж уклонился от атаки.

Способности интеллекта находятся в диапазоне от 0 и выше. Интеллект определяет шансы того, что заклинание достигнет своей цели. У заклинания есть собственные шансы на успех, но когда вы добавляете способности интеллекта, шансы увеличиваются. Если заклинатель имеет значение интеллекта 100, у заклинания на 100 процентов больше шансов сработать.

Сопrotивляемость — это способность персонажа уменьшать ущерб, наносимый заклинаниями. Значение находится в диапазоне от 0 до 100 процентов. Процент применяется к количеству наносимых заклинанием повреждений и результат используется как итоговое значение наносимого персонажу ущерба.

И, наконец, меткость — эта способность измеряется числом в диапазоне от 0 до 999. Снова случайное число сравнивается со значением меткости; если случайное число больше значения способности, значит атака не попала в цель.

Помимо способностей в описании персонажа должна присутствовать и другая информация — например, здоровье игрока. Здесь необходимо знать несколько дополнительных деталей о каждом персонаже, чтобы определить, насколько он здоров и силен. Встречайте атрибуты персонажа.

## **Атрибуты персонажа**

Атрибуты персонажей похожи на способности, за исключением того, что атрибуты определяют различные параметры персонажа. Физическое здоровье — это атрибут; от его изменения зависит, сколько ран может выдержать персонаж. Эти раны могут быть вылечены и в таком случае здоровье персонажа увеличивается.

В игре требуется всего несколько атрибутов. В таблице 12.2 описаны четыре атрибута, которые я буду использовать в этой главе.



**Таблица 12.2.** Атрибуты персонажа

<b>Атрибут</b>	<b>Описание</b>
Здоровье	Здоровье персонажа может находиться в диапазоне от нуля и выше. Чем больше значение, тем больше ранений может вынести персонаж. Нулевое значение означает, что персонаж мертв.
Мана	Мана определяет количество магической энергии, имеющейся в запасе у персонажа. Каждое заклинание требует определенного количества маны, расходуемого при каждом произнесении.
Очки опыта	Думайте о назначенном числе, как о сумме опыта, который вы приобрели в жизни. По мере увеличения опыта персонажа в игре он становится сильнее и больше знает. Опыт персонажей измеряется в числовой форме.
Уровень опыта	Достаточно часто способности и атрибуты персонажей увеличиваются. Уровни опыта — это ступени увеличения. Уровень опыта определяется по количеству очков опыта, как будет показано позже в разделе «Увеличение опыта и силы».

Здоровье персонажа измеряется в очках здоровья (health points, HP), а мана измеряется в очках маны (mana points, MP). MP нужны персонажу для произнесения заклинаний. Каждый раз при произнесении заклинания тратится определенное количество MP; если маны недостаточно, заклинание не может быть произнесено.

Опыт и уровни опыта позволяют игрокам следить за прогрессом и увеличением силы их персонажей. Опыт относится только к персонажам игроков и обсуждается в разделе «Увеличение опыта и силы» далее в этой главе.

### **Дополнительные статусы персонажа**

Способности и атрибуты замечательно описывают возможности персонажа. Конечно, эти способности и атрибуты описывают персонаж на пике его возможностей. Персонажа могут преследовать болезни, делающие его слабее или сильнее. Персонаж может быть отравлен или зачарован, из-за чего он станет неповоротливым.

Эти дополнительные статусы магические по своей природе и помогают повернуть ход игры в критических точках. В таблице 12.3 перечислены некоторые дополнительные статусы, используемые во многих играх (и в этой книге), и описано действие, производимое ими на персонажа.

Таблица 12.3. Дополнительные статусы и их действие

<b>Статус</b>	<b>Эффект</b>
Отравление	Отравление медленно уменьшает здоровье персонажа, пока не будет принято противоядие. В этой книге отравленный персонаж теряет два очка здоровья каждые четыре секунды.
Сон	Спящий персонаж ничего не может делать, пока не проснется. Разбудить спящего может удар другого персонажа, или придется ждать, пока эффект не пройдет сам (с 4-х процентной вероятностью).
Паралич	Парализованный персонаж ничего не может делать точно так же, как и спящий. Но парализованный персонаж может снова начать двигаться только если кто-то снимет с него заклятье или эффект не пропадет сам (с двухпроцентной вероятностью).
Слабость	Способности к атаке и защите у ослабленного персонажа уменьшаются наполовину.
Усиление	Способности персонажа к атаке и защите увеличиваются на 50 процентов.
Зачарованность	Соппротивление к магии у зачарованного персонажа снижается наполовину.
Барьер	Соппротивляемость защищенного магическим барьером персонажа увеличивается на 50 процентов.
Ошеломление	Ошеломленные персонажи теряют половину интеллектуальных способностей, пока не пройдет ошеломление.
Неуклюжесть	Проворность неуклюжих персонажей уменьшается на 25 процентов; это уменьшает возможность уклоняться от физических атак.
Устойчивость	Твердо стоять на ногах — вот значение этого статуса, и у таких персонажей проворность увеличивается на 50 процентов.
Замедление	Обычно у персонажа есть установленная скорость перемещения (измеряемая в блоках за секунду), но у замедленных персонажей эта скорость наполовину уменьшается.
Ускорение	Увеличивает скорость перемещения персонажа на 50 процентов.
Слепота	У ослепленного персонажа на 25 процентов больше шансов промахнуться мимо цели при атаке.
Орлиный глаз	Персонажи с орлиным глазом на 50 процентов чаще попадают в цель при атаке.
Немота	Немые персонажи не могут произносить заклинания, пока заклинание не будет снято.

Как видите, отдельные статусы помогают персонажам, а не мешают им. Позднее в разделах «Функции заклинаний» и «Использование правил битвы для атаки» вы увидите как внедрить эти статусы в ваш проект.

### ***Классы персонажей***

Персонажи бывают всех форм и размеров — большие, маленькие и даже белые и пушистые. Факт в том, что некоторые персонажи имеют различные атрибуты, делающие их особенными.

Для каждого отдельного типа персонажей в вашей игре есть класс персонажей. Вы можете думать о классах персонажей как о способе распределения персонажей вашей игры. Если взглянуть глубже, классы могут определять конкретные типы персонажей в деталях.

Классифицировать персонажа только как человека недостаточно. Вместо этого подходящей классификацией в ряде случаев могла бы быть — человек-воин. Это применимо ко всем персонажам. Например, вы можете классифицировать дракона как ледяного дракона, огненного дракона, каменного дракона и т.д.

Причины необходимости определения класса коренятся в проекте вашей игры (обратитесь к главе 11, «Определение и использование объектов», за дополнительной информацией об использовании классов по отношению к предметам и персонажам). Предметам в вашей игре назначена переменная использующего их класса, состояние которой определяет, какие классы могут воспользоваться конкретным предметом. Широкий меч может быть на вооружении только у персонажа, являющегося человеком-воином или гномом, а свиток с заклинаниями может использовать только класс магов.

Классы персонажей вступают в игру также при обработке сражений. Отдельные виды атак, будь они физические или магические, могут сильнее или слабее действовать на отдельные классы персонажей. Возьмем, к примеру, заклинание огненного шара. Поскольку оно базируется на огне, оно может наносить больший ущерб ледяным монстрам, чем огненным (фактически, оно может даже лечить огненных монстров).

Классы персонажей назначаются по номерам и полностью зависят от вашего проекта.

### ***Действия персонажей***

С каждым персонажем вашей игры связан набор действий, которые этот персонаж может выполнять, а с каждым действием связана анимация, воспроизводимая на экране. Размахивание оружием, произнесение заклинаний или разговор с другим персонажем — все это действия и ваша задача определить, какие именно действия смогут выполнять персонажи вашей игры.

Каждое действие в игре производит какой-то эффект. Ходьба перемещает персонаж, а атака приводит к выхватыванию оружия и

определению кого или что надо ударить. Внутри программы имеет место только эффект; вне ее игрок видит графическую анимацию для представления действия. В таблице 12.4 показаны действия, которые я реализую в этой книге.

**Таблица 12.4.** Действия персонажа

<i><b>Действие</b></i>	<i><b>Описание</b></i>
Атака	Размахивая оружием, персонаж набрасывается на другого персонажа, находящегося перед ним. В этой книге реализован только один тип атаки, но в своей игре вы можете добавить другие типы атак.
Произнесение заклинания	Демонстрируется древнее искусство магии, когда персонаж выполняет небольшие ритуалы с целью вызвать разрушающее или исцеляющее заклинание.
Перемещение	Ходьба, полет, бег — все это способы перемещения по вашему игровому миру. Вашему персонажу необходим стандартный метод передвижения, и в этой книге таким методом будет ходьба.
Ожидание	Когда персонаж стоит, он находится в ожидании. Персонаж может выглядеть скучающим, настороженным или постоянно озираясь вокруг, но, независимо от того что он делает, его считают ожидающим.
Использование предмета	Когда персонаж решает начать использовать какой-нибудь из собранных им предметов, начинается действие этого предмета.
Разговор	Общение с игровыми жителями — это действие разговора. Игроки хотят, чтобы их персонажи не стояли при этом неподвижно, а показывали, как они взаимодействуют друг с другом — двигали руками, шевелили губами или что-нибудь другое.
Ранение	Персонаж, задетый атакой, неважно магической или физической, обычно требует несколько секунд на восстановление. Период восстановления называется действием ранения и в это время игрок не может выполнять никакие другие действия.
Смерть	Получив достаточное количество повреждений персонаж умирает. Однако недостаточно просто умертвить его; удаление персонажа из игры сопровождается драматической анимацией гибели.

Некоторые действия могут выполняться только в заданное время. Возможно, персонажи не могут атаковать во время последовательности перемещений, оставляя боевые действия для отдельной боевой последовательности. Даже когда эти действия могут исполняться возможно наличие других факторов, ограничивающих возможность выполнения некоторых действий.

Возьмем, к примеру, действие атаки. Предположение, что это действие может выполняться в любой момент игры, не означает, что игрок может сидеть и непрерывно рубить. Вы используете ограничивающий фактор, называемый *перезарядкой* (*charge*), который задает временную задержку между атаками. Фактически, все действия за исключением ожидания и перемещения не работают во время перезарядки персонажа.

Когда выполняется одно из таких действий, значение заряда персонажа сбрасывается. Затем постепенно выполняется перезарядка, пока персонаж снова не будет в состоянии выполнить другое действие. Скорость увеличения заряда индивидуальна для каждого типа персонажа.

Хотя каждый персонаж игры может выполнять действия, показанные в таблице 12.4, определенные действия могут выполнять только отдельные типы персонажей. Например, только персонажи игроков могут использовать предметы, для врагов это действие недоступно.

Чтобы лучше понять, что каждый тип персонажей может выполнять, давайте более пристально рассмотрим каждый из типов.

## **Персонажи игроков**

Мир вращается вокруг ваших персонажей игроков (PC), поэтому значительная часть разработки игры связана с управлением ими. У персонажей игроков больше всего действий и для них открыто больше всего возможностей. Кроме ранее упомянутых действий персонажей, у PC есть возможность управлять их игровыми ресурсами, в которые входят предметы и магические заклинания. Кроме того, идет построение персонажа, чтобы сделать его более сильным.

В следующих разделах показан каждый из этих аспектов, вступающих в игру при работе с персонажами игрока.

### ***Передвижение игрока***

Наиболее важная способность персонажа игрока — передвижение. В огромном мире значительная часть удовольствия состоит в обследовании всех его укромных уголков. Вы должны дать персонажам игроков как можно больше способов перемещения по миру — ходьба, полет, плавание, телепортация и т.д.

И снова, если вернуться к основам, наиболее важный способ перемещения по миру — ходьба. Каждому персонажу вашей игры назначено значение, определяющее скорость его передвижения, и для игрока, чем быстрее движется персонаж, тем лучше.

### ***Управление ресурсами***

Ресурсы — это предметы и объекты, разбросанные по вашему миру. Они делают игру увлекательной.

В главе 11 было рассмотрено создание предметов, но вопросы их использования не были полностью раскрыты. Фактически, предметы

бесполезны, если нет кого-то, кто мог бы воспользоваться ими, так что сейчас самое время рассмотреть, как игроки взаимодействуют с этими предметами. Эликсиры можно выпивать, оружие и броню носить, а золото тратить — и все это относится к тому, как проектировать эти предметы (глава 11) и как игроки могут использовать их!

Ресурсы это не только предметы, но и магические заклинания. Заклинания исключительно полезные инструменты, и чтобы сделать что-нибудь в мире ваш персонаж должен изучить столько заклинаний, сколько возможно. Как игрок учит используемые заклинания? Через непрерывное увеличение опыта!

## **Увеличение опыта и силы**

В типичной ролевой игре персонажи могут увеличивать свой опыт — каждое найденное сокровище и каждая выигранная битва увеличивают их способности. Думайте об опыте, как о числе, и чем больше число, тем мощнее ваш персонаж. Когда достигается заданный уровень опыта, персонаж получает дополнительные преимущества.

Например, предположим, что игровой персонаж по имени Джордж только что убил своего тысячного ога. На протяжении своего боевого пути он становился все более и более сильным. Теперь своим мечом он может наносить в три раза больше повреждений, чем в начале приключения. Его физическая сила увеличилась, он стал проворнее и у него больше шансов увернуться от атаки. Увеличились его интеллектуальные способности, и он изучил несколько новых мощных заклинаний.

У персонажей есть *очки опыта* (*experience points*) и *уровень опыта* (*experience level*). Каждый эпизод увеличивает количество очков опыта персонажа. Через заданные интервалы очков опыта у персонажа повышается уровень опыта. При повышении уровня опыта персонаж получает дополнительные преимущества, которые обычно включают увеличение способностей и заклинания.

В этой книге очки опыта персонажи получают за убийство врагов. Количество получаемых очков закодировано в описаниях персонажей. Убийство глупого мелкого демона может дать вашему персонажу ничтожные 10 очков опыта, в то время как победа над красным драконом двухсотого уровня принесет персонажу колоссальные 20 000! Вы определяете количество опыта, извлекаемого из убийства врагов.

Установка уровней опыта и получаемых преимуществ — это часть работы проектировщика. Персонажу первого уровня может потребоваться 500 очков опыта, чтобы достичь уровня 2, а переход на третий уровень может потребовать 1 200 очков опыта.

Определяя количество очков опыта, необходимое для конкретного уровня, вы можете использовать в качестве критерия повышения уровня среднее количество опыта, получаемого от убийства врагов в конкретной области игры. Например, если ваш персонаж находится на «демонической

пустоши», каждый убитый демон приносит ему 10 очков опыта, и вы хотите, чтобы для перехода на второй уровень игрок убил не менее 20 монстров, это означает, что уровень 2 требует 200 очков опыта.

Помимо дополнительных преимуществ, с увеличением уровня опыта увеличиваются способности и атрибуты. У персонажа увеличиваются максимальные значения очков здоровья и маны; игрок становится сильнее и может нанести больше повреждений. Произносить заклинания становится легче, и с увеличением количества очков маны ваш персонаж может произнести больше заклинаний, прежде чем его силы иссякнут. Что на самом деле будет происходить с персонажем игрока при увеличении уровня, определяется проектом игры. Я не буду слишком углубляться в этот вопрос (но вы увидите, как я увеличиваю способности персонажа при увеличении количества очков опыта в примере игры из главы 16, «Объединяем все в законченную игру»).

## Независимые персонажи

Вы обращаетесь с независимыми персонажами во многом так же, как и с персонажами игроков, за исключением того, что они полностью находятся под управлением движка игры. Это различие создает некоторые ограничения для проекта и программные ситуации, и вы должны разработать систему искусственного интеллекта, которая сможет подражать осмысленному поведению.

Пересмотрев относящиеся к персонажам аспекты, вы увидите, что для NPC требуется достаточно мало управления:

- **Перемещение.** NPC должны перемещаться по игровому миру, хотя и не так сложно как PC. Это достаточно просто реализовать с помощью скриптов, но остается проблема препятствий ландшафта и взаимодействия с персонажами игроков.
- **Общение.** И снова, хотя и не на том же уровне, что PC, NPC должны действовать «по-человечески», и часть этих действий — использование навыков общения.
- **Сражения.** Персонажи игроков сражаются в основном с NPC, так что последние должны быть достойными противниками. Даже у самого непритязательного NPC есть навыки самосохранения, и он должен продемонстрировать их при появлении смертельной угрозы.

Другие важные аспекты управления, такие как управление ресурсами, неприменимы к NPC. Предметы, которые несут NPC, записаны в проекте персонажа. Изредка у них есть свобода выбора, что нести или использовать. Также, поскольку игрок не управляет NPC, у них есть разработанная «цельная жизнь», поэтому развитие персонажа не влияет на них.

Мы пока пропустим сражения, поскольку они относятся только к врагам (а не ко всем NPC), и обратим внимание на перемещение и общение, оказывающие значительный эффект на NPC. Эти два действия требуют правдоподобного искусственного интеллекта, чтобы персонажи в игре стали

разумными. На самом деле NPC достаточно только выглядеть разумными. Подо всеми внешними наслоениями они следуют небольшому набору инструкций и правил, управляющих их действиями.

Например, созданная Sega популярная игра Phantasy Star Online использует минимум взаимодействий с NPC. Есть несколько лавочников, продающих предметы, банк для хранения денег и других предметов, лечащие раны целители и гильдии, отправляющие вас на поиски. Игровые монстры по многим стандартам просто тупы; они бродят вокруг, пока не увидят персонажа игрока, а затем бросаются на него в атаку. Такой вариант управления NPC замечательно подходит для быстроидущих ролевых игр, но недостаточно хорош для более серьезных игроков.

С другой стороны, такие игры, как Ultima Online от Origin заполнены NPC. Как сравнить эти две игры? NPC в Ultima Online достаточно ограничены в терминах их искусственного интеллекта — они могут только ходить вокруг, следовать за персонажем, оставаться неподвижными, атаковать, лечить, охранять и действовать как банкиры и лавочники. Настоящая магия Ultima Online — используемые NPC скрипты. У каждого NPC может быть присоединенный скрипт, позволяющий NPC выполнять дополнительные действия. Например, разговор с NPC не приведет ни к чему важному, пока игрок не даст NPC специальный предмет, в результате чего NPC завещает игроку магический меч.

Каким путем пойти? Если вы хотите сверхсложный искусственный интеллект, пусть будет так, но ваш игровой мир должен быть буквально напичкан сотнями NPC, и чем проще с ними обращаться, тем лучше.

В этой книге я моделирую схему управления NPC из Ultima Online. NPC назначаются определенные движения, и для них разрешено использовать скрипты. Для простоты PC должны взаимодействовать с NPC перед активацией скрипта.

## **Персонажи монстров**

Монстры — это скрытые NPC. MC программируются точно так же, как NPC, но обладают двумя дополнительными формами интеллекта — охотой и атакой персонажей игроков.

Монстры могут ходить вокруг, следовать за кем-то, укрываться, патрулировать по маршруту и даже стоять неподвижно — точно так же, как и обычные NPC. Однако, когда персонаж игрока попадает в пределы досягаемости атаки MC (физической или магической), начинается игра по правилам. Враг будет последовательно атаковать ближайшего PC, пока этот персонаж не умрет или не покинет пределы досягаемости атак монстра.

У монстров есть инстинкт самосохранения. Если монстр потерял половину очков здоровья и у него есть возможность произнести исцеляющее заклинание, он попытается вылечиться. Аналогично, если монстр заболел, он попытается исправить положение. И, наконец, монстр может попробовать



улучшить свои параметры, применяя на себя заклинания для установки дополнительных статусов.

Монстры — это единственный тип персонажей, на которые игрок может напасть (или должен быть в состоянии напасть). Поскольку РС получает опыт только убивая врагов, вы должны обеспечить наличие в вашей игре необходимого количества монстров.

## **Графика персонажей**

До этого момента мы обсуждали только функциональные возможности персонажа. Действительность заключается в том, что значение имеют только функциональные возможности, но игрокам необходимо какое-нибудь визуальное представление их альтер-эго. Благодаря мощи программ трехмерного моделирования и графическому ядру у вас не будет проблем, касательно графических аспектов ваших персонажей.

Каждый персонаж вашей игры может выполнять конкретные действия. Эти действия (по крайней мере, те, которые используются в этой книге) следующие: ходьба, ожидание, взмах оружием, произнесение заклинания, ранение и смерть. У каждого из этих действий есть отдельная анимация, которая используется совместно с сеткой для создания графического компонента персонажа.

Как вы узнаете в последующем разделе, «Создание класса управления персонажем», сетка каждого персонажа загружается в объект **cMesh**, а каждая анимация загружается в объект **cAnimation**. Эти два объекта графического ядра уникальны для каждого персонажа в вашей игре (несколько экземпляров одного и того же персонажа используют только одну сетку и объект анимации). Ваша задача — загрузить эти сетки и анимации и визуализировать персонажи на экране в каждом кадре.

---

### **ПРИМЕЧАНИЕ**

*Персонаж игрока (player character)* — это альтер-эго игрока. Независимые персонажи — это персонажи, над которыми игрок не имеет непосредственного контроля. Часто NPC — это горожане, торговцы, банкиры, лекари и монстры. Да, даже монстры являются NPC, но в этой книге я провожу четкое различие между NPC и MC. NPC не атакуют игрока, а MC — атакуют.

---

## **Перемещение персонажей**

Теперь, когда вы описали персонажей, пришло время поместить их в мир и заставить перемещаться по нему. Здесь вам потребуются отдельные системы управления для РС и NPC. РС являются персонажами, которыми непосредственно управляют игроки, и у них больше возможностей.

Если вы смотрели демонстрационные программы к главам 8 и 9 с прилагаемого к книге CD-ROM, то уже знакомы с созданной мной системой управления игроком (которую я предпочитаю называть непосредственным управлением). Эти демонстрационные программы позволяют вам

перемещать персонажа игрока, используя клавиши управления курсором и мышью.

Для демонстрационных программ с видом от первого лица (таких, как **NodeTree** из главы 8), нажатие на клавишу перемещения курсора вверх передвигает персонаж вперед, нажатие на клавишу перемещения курсора вниз передвигает игрока назад и нажатие на клавиши перемещения курсора вправо и влево перемещают игрока вправо и влево соответственно. Передвижение мыши вращает точку просмотра. Для демонстрационных программ с видом от третьего лица (таких, как **3Din2D** из главы 9), клавиши управления курсором перемещают персонаж в соответствующем направлении (нажмите клавишу перемещения курсора вверх для перемещения вверх, клавишу перемещения курсора влево для перемещения влево и т.д.).

---

**ПРИМЕЧАНИЕ**

Термины «вид от первого лица» и «вид от третьего лица» описывают местоположение точки зрения игрока. Для игр с видом от первого лица точка зрения игрока находится в глазах персонажа. Для игр с видом от третьего лица используется другая точка зрения — сзади, сбоку или другая перспектива.

---

При работе с РС редко применяется что-либо более сложное, чем эти системы управления. Однако, вещи становятся более сложными, когда игрок начинает ходить, сталкиваясь с другими персонажами и объектами (такими, как двери). Вы уже видели, как выполняется обнаружение столкновений, так что это не должно составить для вас большой проблемы.

Все становится несколько более сложным для NPC. Больше нет игрока, отвечающего за перемещение по карте; в дело вступает движок игры. Вы можете позволить NPC ходить вокруг по простым направлениям, но вместо того, чтобы перемещаться как РС, они руководствуются набором базовых правил перемещения:

- Остановка.
- Бесцельное блуждание по уровню или его заданной области.
- Хожение по заданному маршруту.
- Следование за персонажем.
- Уклонение от персонажа.

Предшествующего списка действий вполне достаточно для начала, и так как надо начинать с чего-нибудь, начнем с наиболее логичного места — управления персонажем игрока.

## Управление персонажем игрока

Игрок — это самый главный персонаж игры, так что необходим полный контроль над ним. Обычно игры используют простые схемы

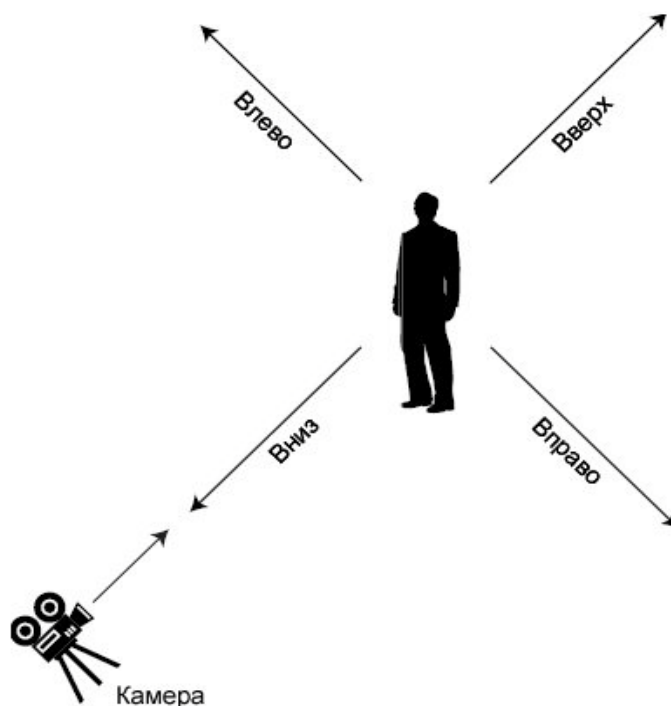
непосредственного управления, подобные той, о которой вы читали в разделе «Передвижение игрока».

Схемы непосредственного управления позволяют вам вручную перемещать персонажа вперед, назад, влево, вправо, вверх и через карту. Ничто не может быть более очевидным, так что в этой книге я предпочитаю использовать для управления персонажами методы непосредственного управления.

Есть два типа непосредственного управления — управление направлением и управление поворотом, и о них я расскажу в следующих подразделах.

### **Управление направлением**

Управление направлением использует элементы управления, такие как клавиши управления курсором или джойстик, для перемещения персонажа в конкретном направлении. Управление направлением лучше подходит для игр с видом от третьего лица, где камера показывает персонажа сзади или сбоку (как показано на рис. 12.1).



**Рис. 12.1.** При управлении направлением персонаж перемещается в направлении, определяемом нажатой клавишей. Клавиша перемещения курсора вверх перемещает персонаж на экране вверх (от камеры), клавиша перемещения курсора влево перемещает персонаж влево и т.д. Этот метод управления при перемещении персонажа принимает во внимание местоположение камеры, так что клавиша перемещения курсора вверх всегда перемещает персонаж вверх (вдаль), независимо от угла вида камеры

Для перемещения персонажа при использовании управления направлением вам необходимо знать угол наклона камеры (относительно оси Y), местоположение персонажа и направление, в котором персонаж должен перемещаться (0.0 означает вверх или от камеры, 1.57 означает вправо, 3.14 означает вниз и 4.71 означает влево).

Предположим, что координаты персонажа представлены парой переменных:

```
float XPos, ZPos; // YPos = высота, она не требуется
```

Угол наклона камеры (относительно оси Y) представляется переменной:

```
float CameraYAngle; // Угол вида
```

И, наконец, у вас есть направление и расстояние, на которое вы хотите переместить персонаж:

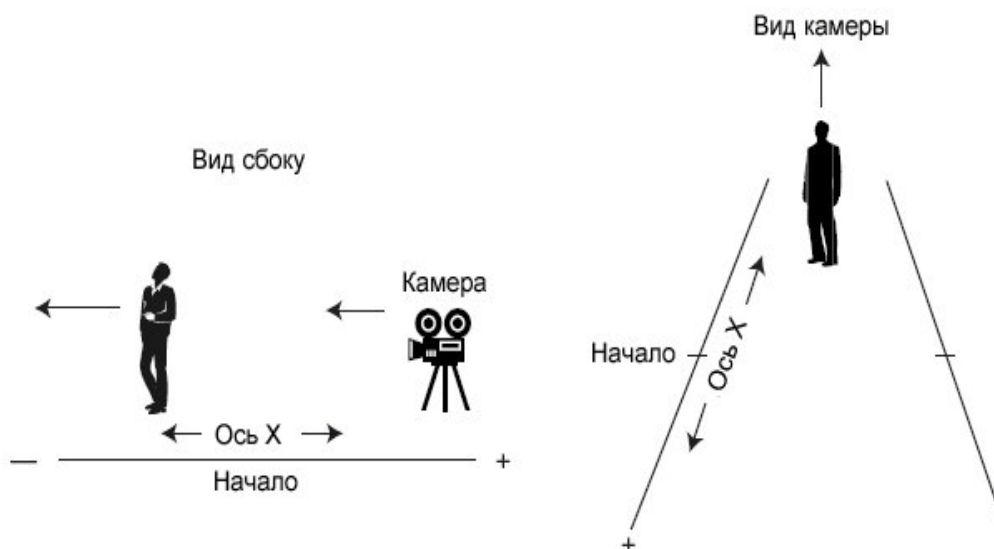
```
float Direction; // Направление перемещения
float Distance; // Как далеко переместиться
```

---

**ПРИМЕЧАНИЕ** Использование значения расстояния может вызвать проблемы, поскольку вы не контролируете скорость обновления кадров в вашем движке. Когда игра работает на компьютере с полной скоростью и обновляется 100 раз в секунду, игровой персонаж может перемещаться гораздо быстрее, чем на компьютере с частотой обновления 30 раз в секунду. Вам необходимо ограничить количество обновлений игры в каждом кадре, как я делаю позже в разделе «Использование cCharacterController» и главе 16.

---

Затем создадим пример сцены, в котором персонаж находится в начале координат и хочет переместиться вверх (относительно вида камеры); камера располагается справа от начала координат вдоль оси X и направлена влево, как показано на рис. 12.2.



**Рис. 12.2.** Камера направлена влево, а персонаж перемещается вверх

Вот устанавливаемые переменные и проводимые вычисления:

```
XPos = ZPos      = 0.0f; // Местоположение персонажа
CameraYAngle     = -1.57f; // Угол камеры
Direction        = 0.0f; // Направление перемещения
float Distance = 4.0f; // Расстояние перемещения
```

```
// Новое направление (угол) для перемещения
float NewAngle = CameraYAngle + Direction - 1.57f;

// Перемещение персонажа
XPos += (float)cos(NewAngle) * Distance;
ZPos += (float)-sin(NewAngle) * Distance;
```

К этому моменту персонаж переместился на 4 единицы от камеры, а значит он сейчас находится в точке  $X = -4.0$ ,  $Z = 0.0$ , куда вы и хотели его поместить. Остается только задача определения высоты персонажа (для чего используется трехмерный движок).

В этом нет никаких проблем. В главе 8 мы исследовали как использовать соответствующие проверки пересечения, чтобы определить высоту точки полигона, которая будет использоваться как координата Y персонажа. Глава 8 описывает применение функции **GetHeight** для определения координаты Y, необходимой для размещения персонажей в трехмерном пространстве.

### Управление поворотом

Управление поворотом позволяет игроку поворачивать персонаж, используя клавиши перемещения курсора вправо и влево и передвигать его вперед и назад с помощью клавиш перемещения курсора вверх и вниз.

В ряде аспектов управление поворотом лучше чем управление направлением, поскольку вычисления перемещения более просты. Персонажу здесь нужно хранить значение направления, однако, оно представляет направление, в котором обращено лицо персонажа (0.0 означает направление вдоль положительного направления оси Z, 1.57 — вдоль положительного направления оси X, 3.14 — вдоль отрицательного направления оси Z и 4.71 — вдоль отрицательного направления оси X). Подразумевается, что используется следующая переменная направления:

```
float Facing = 0.0f; // Направление лица персонажа
```

Теперь вы можете сказать, что задействование левого элемента управления (нажатие клавиши перемещения курсора влево или наклон джойстика влево) поворачивает персонаж влево (отрицательное вращение), а задействование правого элемента управления поворачивает персонаж вправо (положительное вращение):

```
// Используем ядро ввода:
// Keyboard = ранее инициализированный
//           объект cInputDevice для клавиатуры
Keyboard.Read();

// Поворот влево?
if(Keyboard.GetKeyState[KEY_LEFT] == TRUE)
    Facing -= 0.01f; // Поворот влево на .01 радиан

// Поворот вправо?
if(Keyboard.GetKeyState[KEY_RIGHT] == TRUE)
    Facing += 0.01f; // Поворот вправо на .01 радиан
```

С учетом угла из переменной **Facing** перемещение вперед и назад выглядит так:

```
// Перемещение вперед?
if (Keyboard.GetKeyState[KEY_UP] == TRUE) {
    XPos += (float)cos(Facing-1.57f) * Distance;
    ZPos += (float)-sin(Facing-1.57f) * Distance;
}
// Перемещение назад?
if (Keyboard.GetKeyState[KEY_DOWN] == TRUE) {
    XPos += (float)-cos(Facing-1.57f) * Distance;
    ZPos += (float)sin(Facing-1.57f) * Distance;
}
```

## Управление от первого лица

Последний тип непосредственного управления я использую для игр с видом от первого лица, где вы смотрите на мир через глаза персонажа. В этой форме управления клавиши управления курсора используются для перемещения персонажа влево, вправо, вперед и назад, а мышь для поворота вида (также как вы вертите головой, чтобы осмотреться вокруг).

Нажатие кнопки перемещения курсора вверх перемещает персонаж вперед в направлении его взгляда, а нажатие клавиши перемещения курсора вниз передвигает персонаж назад. Клавиши перемещения курсора вправо и влево перемещают персонаж вправо и влево. Управление от первого лица похоже на управление поворотом, о котором вы только что читали, но в управлении от первого лица мышь поворачивает персонаж.

Однако в данном случае поворачивается не персонаж, а камера (поскольку вид камеры представляет вид из глаз персонажа). Это добавляет пару новых переменных, представляющих углы камеры:

```
float XAngle = 0.0f, YAngle = 0.0f; // Углы просмотра персонажа
```

Две показанные переменные хранят углы просмотра, которые модифицируются, когда игрок перемещает мышь. Вот код для изменения этих углов:

```
// Подразумевается использование ядра ввода
// Mouse = ранее созданный cInputDevice для мыши
// То же относится и к клавиатуре
Mouse.Read();
Keyboard.Read();

// Поворот персонажа на основе угла мыши
XAngle += Mouse.GetYDelta() / 200.0f;
YAngle += Mouse.GetXDelta() / 200.0f;

// Перемещение персонажа
if (Keyboard.GetKeyState[KEY_UP] == TRUE) {
    XPos += (float)cos(YAngle-1.57f) * Distance;
    ZPos += (float)-sin(YAngle-1.57f) * Distance;
}
if (Keyboard.GetKeyState[KEY_DOWN] == TRUE) {
    XPos += (float)-cos(YAngle-1.57f) * Distance;
```

```
ZPos += (float)sin(YAngle-1.57f) * Distance;
}
if(Keyboard.GetKeyState[KEY_LEFT] == TRUE) {
    XPos += (float)cos(YAngle-3.14f) * Distance;
    ZPos += (float)-sin(YAngle-3.14f) * Distance;
}
if(Keyboard.GetKeyState[KEY_RIGHT] == TRUE) {
    XPos += (float)cos(YAngle) * Distance;
    ZPos += (float)-sin(YAngle) * Distance;
}
```

Заметьте, что всякий раз когда пользователь перемещает мышь, для поворота вида используется значение приращения (количество движения). После этого легко вычислить направление, в котором перемещать персонаж.

## Управление независимыми персонажами

Как вы можете предположить после прочтения предыдущих разделов, управлять игроком относительно просто. Теперь начинается трудная часть — управление игровыми NPC. Этот раздел покажет вам различные методы перемещения ваших игровых NPC.

Хотя игры могут обманом завлечь вас в размышления о сложных схемах перемещения NPC по миру, это не так.

Помните упомянутые ранее пять общих типов перемещения NPC — ожидание, блуждание в заданной области, ходьба по маршруту, следование за персонажем и уклонение от персонажа? Держа их в уме, вы можете пристальнее взглянуть на ваши любимые ролевые игры, чтобы определить, какие схемы управления они используют.

Что касается вашей ролевой игры, уделите момент для исследования следующих схем управления и их реализации.

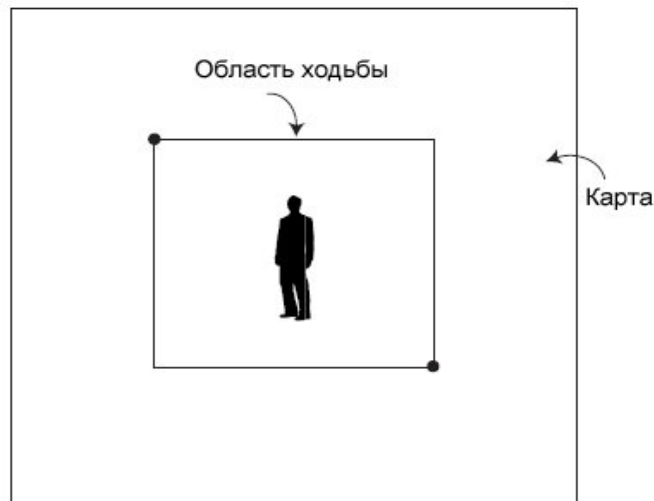
### **Ожидание**

Здесь не о чем думать — просто разместите персонаж, и он будет стоять, оставаясь обращенным в заданном направлении. Направление здесь — угол поворота.

### **Блуждание по области**

Такие игры, как Ultima Online позволяют NPC бродить по заданной области, которая может быть целым уровнем или задаваемой вами его отдельной частью. Чтобы сохранить вещи простыми, вы можете задать диапазон, в котором хотите разрешить блуждания персонажа, в виде заданного диапазона координат (как показано на рис. 12.3). Эти координаты хранятся в следующих переменных:

```
float WanderMinX, WanderMinY, WanderMinZ;
float WanderMaxX, WanderMaxY, WanderMaxZ;
```



**Рис. 12.3.** Бродя по округе персонажу нужно знать ограничения. Задав небольшую область на карте (как показано здесь) вы ограничиваете перемещения персонажа

Теперь, подразумевая, что вы отслеживаете координаты персонажа в трех переменных, вы можете случайным образом перемещать его вокруг, проверяя допустимость этих перемещений:

```
// Координаты персонажа
float CharXPos, CharYPos, CharZPos;

// Дальность перемещения - пропущено перемещение YMove
float XMove, ZMove;

// Расстояние, на которое перемещается персонаж
float Distance;

// Определяем случайное направление перемещения,
// продолжаем цикл пока не найдем
while(1) {
    float Direction = 6.28f / 360.0f * (float)(rand() % 360);
    XMove = cos(Direction) * Distance;
    ZMove = sin(Direction) * Distance;

    // Проверяем допустимо ли перемещение, игнорируя высоту
    if(CharXPos+XMove >= WanderMinX &&
        CharXPos+XMove <= WanderMaxX &&
        CharZPos+ZMove >= WanderMinZ &&
        CharZPos+ZMove <= WanderMaxZ) {

        // Перемещение разрешено, обновляем координаты
        CharXPos += XMove;
        CharZPos += ZMove;
        break; // Выходим из цикла
    }
}
```

#### ПРИМЕЧАНИЕ

Не стоит случайным образом перемещать персонаж в каждом кадре, или ваши персонажи будут выглядеть, будто у них истерика. Вместо этого обновляйте направление движения персонажа только раз в несколько секунд или около этого.



## Ходьба по маршруту

Хотя у NPC недостаточно интеллекта, чтобы знать, как они могут ходить по уровню, вы можете назначать им маршруты для путешествий. Эти маршруты включают координаты, которые должны быть достигнуты, прежде чем отправиться к следующим координатам. Когда достигнут последний набор координат, персонаж возвращается к начальным координатам и повторяет весь свой путь снова.

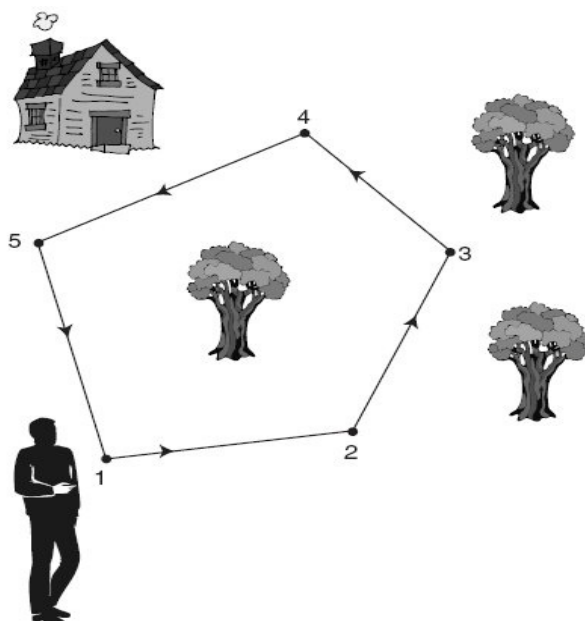
### Использование маршрутных точек

Маршрутные точки определяются как набор координат, и чтобы сохранить привычные трехмерные концепции, можно для их хранения использовать следующую структуру:

```
typedef struct sRoutePoint {
    float XPos, ZPos;
} sRoutePoint;
```

Для создания маршрута вы выбираете точки, которые должен пройти персонаж и формируете массив структур **sRoutePoint** для хранения координат. На рис. 12.4 показана простая карта с пятью отмеченными точками.

```
sRoutePoint Route[5] = {
    { -200.0f, -100.0f },
    { 100.0f, -300.0f },
    { 300.0f, -200.0f },
    { 200.0f, 100.0f },
    { 0.0f, 400.0f }
};
long NumRoutePoints = 5; // Чтобы проще было узнать количество точек
```



**Рис. 12.4.** На воображаемой карте показано пять маршрутных точек. Персонаж начинает с точки 1 и идет прямо к точке 2, а затем к точкам 3 и так до тех пор, пока не достигнет точки 5. После этого персонаж возвращается к точке 1 и начинает обход маршрута снова

## Ходьба от точки к точке

Чтобы переместиться от точки к точке, при ходьбе персонажа по маршруту необходимо сравнивать его текущие координаты с координатами точки, куда он стремится. Вы используете эти координаты совместно со скоростью ходьбы персонажа, чтобы вычислить пару переменных перемещения, обновляющих местоположение персонажа.

Начнем с предположения, что координаты персонажа хранятся в следующих переменных (вместе со скоростью перемещения персонажа):

```
float CharXPos, CharZPos; // Координата Y не нужна
float WalkSpeed; // Скорость ходьбы за кадр
```

Далее предположим, что вы уже установили координаты, в которые хотите переместить персонаж, и поместили их в другую пару переменных:

```
float RouteXPos, RouteZPos; // Снова без координаты Y
```

Теперь начинаем перемещение персонажа, вычисляя переменные передвижения:

```
// Вычисляем расстояние от персонажа до маршрутной точки
float XDiff = (float)fabs(RouteXPos - CharXPos);
float ZDiff = (float)fabs(RouteZPos - CharZPos);
float Length = sqrt(XDiff*XDiff + ZDiff*ZDiff);

// Вычисление перемещения к пункту назначения
float MoveX = (RouteXPos - CharXPos) / Length * WalkSpeed;
float MoveZ = (RouteZPos - CharZPos) / Length * WalkSpeed;
```

Теперь, всякий раз, когда вы обновляете местоположение персонажа в кадре, вам необходимо прибавить **MoveX** и **MoveZ** к координатам персонажа:

```
CharXPos += MoveX;
CharZPos += MoveZ;
```

Оставив это в стороне, вернемся назад и посмотрим, как отслеживать, к какой маршрутной точке направляется персонаж. Когда достигнута одна из маршрутных точек, персонаж должен направиться к следующей. Чтобы определить, достигнута ли маршрутная точка, вы определяете расстояние от персонажа до нее; если расстояние находится в заданных пределах, персонаж достиг точки и можно продолжать процесс со следующей маршрутной точкой.

## Быстрее Пифагора

Чтобы определить расстояние от маршрутной точки, вы можете использовать теорему Пифагора, но для ускорения работы вы можете отбросить операцию **sqrt** и использовать вместо нее сумму квадратов длин. Чтобы увидеть, что я подразумеваю, взгляните на следующие две строки кода:

```
float Distance = sqrt(Length1*Length1 + Length2*Length2);  
float Distance = Length1*Length1 + Length2*Length2;
```

---

**ПРИМЕЧАНИЕ**

Теорема Пифагора, возможно, самая известная теорема в геометрии. Она утверждает, что квадрат длины гипотенузы в прямоугольном треугольнике равен сумме квадратов длин катетов. Фактически, это означает, что квадратный корень длин двух сторон (они должны быть возведены в квадрат и сложены) равен длине третьей стороны прямоугольного треугольника. Правда, просто?

---

Обратите внимание, что показанные две строки кода практически идентичны, за исключением того, что во второй строке опущена функция **sqrt**, из-за чего она выполняется гораздо быстрее. Недостаток в том, что вы не получаете точную длину, но это не представляет проблемы.

Например, вы измеряете расстояние между двумя точками, и хотите видеть, меньше ли оно 40. Если координаты этих двух точек 0, 0 и 30, 20, более быстрое вычисление даст вам расстояние 1 300 (поскольку длины двух сторон 30 и 20, соответственно).

Как теперь определить расстояние? Вычислив квадрат (произведение на само себя) расстояния, вот как! Умножив 40 на 40 вы получаете 1 600. Сравнив вычисленное расстояние между точками, 1 300, вы видите, что оно меньше, чем 1 600 и, следовательно, меньше, чем оригинальное расстояние 40, которое вы проверяете.

Вернемся к тому, о чем я говорил ранее, и используем быстрый метод вычисления расстояния, чтобы определить приблизился ли персонаж вплотную к маршрутной точке. Предположим, вы решили считать маршрутную точку достигнутой персонажем, если персонаж находится в заданном количестве единиц от нее. Используя быстрый метод вычисления расстояния, вы можете применить для проверки следующую функцию:

```
BOOL TouchedRoutePoint(float CharXPos, // Координаты персонажа  
                        float CharZPos,  
                        float RouteXPos, // Координаты маршрутной точки  
                        float RouteZPos,  
                        float Distance) // Проверяемое расстояние  
{  
    // Вычисляем квадрат расстояния для проведения быстрой проверки  
    Distance *= Distance;  
  
    // Вычисляем расстояние  
    float XDiff = (float)fabs(RouteXPos - CharXPos);  
    float ZDiff = (float)fabs(RouteZPos - CharZPos);  
    float Dist = XDiff*XDiff + ZDiff*ZDiff;  
  
    // Возвращаем результат  
    if(Dist <= Distance) // Внутри проверяемого диапазона  
        return TRUE;  
  
    return FALSE; // Вне диапазона расстояний  
}
```

При вызове **TouchedRoutePoint** с координатами персонажа, координатами маршрутной точки и проверяемым расстоянием от точки, вы получаете значение **TRUE**, если персонаж находится в пределах **Distance** единиц от маршрутной точки. Возвращаемое значение **FALSE** свидетельствует, что от персонажа до маршрутной точки больше **Distance** единиц.

## Прохождение маршрута

Наконец-то вы можете объединить все вместе и заставить персонаж ходить от одной маршрутной точки к другой. Вот небольшая программа, которая берет пять предварительно определенных маршрутных точек, помещает персонаж в первую из них и заставляет персонаж бесконечно обходить все точки:

```
sRoutePoint Route[5] = {
    { -200.0f, -100.0f },
    {  100.0f, -300.0f },
    {  300.0f, -200.0f },
    {  200.0f,  100.0f },
    {    0.0f,  400.0f }
};
long NumRoutePoints = 5;

// Координаты персонажа и переменные перемещения
float CharXPos = Route[0].XPos;
float CharZPos = Route[0].ZPos;
float MoveX, MoveZ;
float Speed; // Скорость ходьбы персонажа

// Начинаем отслеживание до второй точки
long TargetRoutePoint = 1;
SetupMovement(TargetRoutePoint);
// Бесконечный цикл перемещения
// и проверки достижения маршрутной точки
while(1) {
    // Персонаж находится в пределах маршрутной точки?
    if(TouchedRoutePoint(TargetRoutePoint, 32.0f) == TRUE) {

        // Переходим к следующей маршрутной точке
        TargetRoutePoint++;
        if(TargetRoutePoint >= NumRoutePoints)
            TargetRoutePoint = 0;

        SetupMovement(TargetRoutePoint);
    }
    // перемещаем персонаж
    CharXPos += MoveX;
    CharZPos += MoveZ;
}

// Функция проверки нахождения в пределах маршрутной точки
BOOL TouchedRoutePoint(long PointNum, float Distance)
{
    Distance *= Distance;
    float XDiff = (float)fabs(CharXPos - Route[PointNum].XPos);
    float ZDiff = (float)fabs(CharZPos - Route[PointNum].ZPos);
```

```

float Dist = XDiff*XDiff + ZDiff*ZDiff;
if(Dist <= Distance)
    return TRUE;
return FALSE;
}

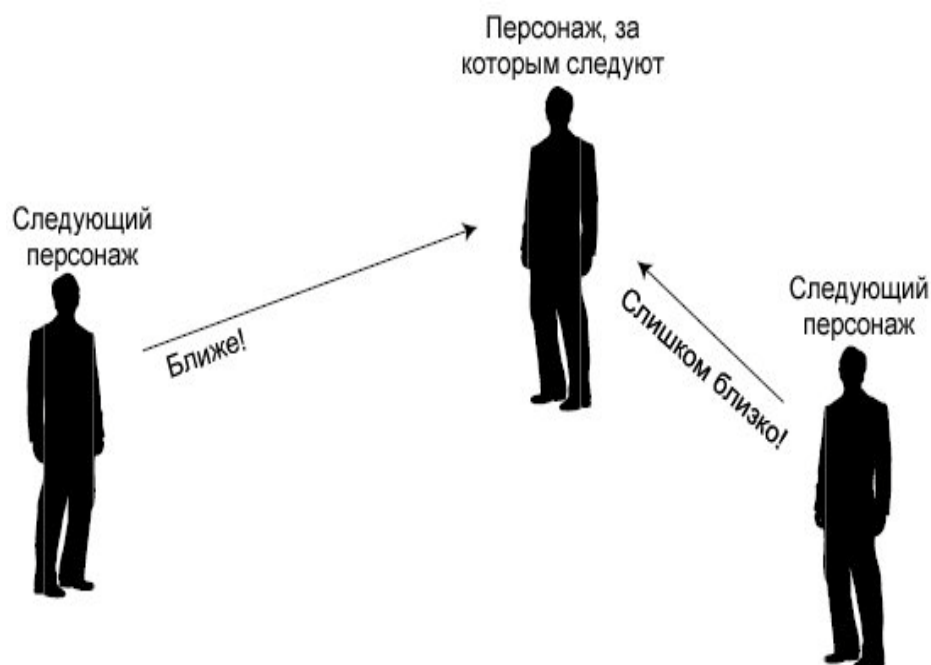
// Функция вычисления переменных перемещения
void SetupMovement(long PointNum)
{
    float XDiff = (float)fabs(CharXPos - Route[PointNum].XPos);
    float ZDiff = (float)fabs(CharZPos - Route[PointNum].ZPos);
    float Length = sqrt(XDiff*XDiff + ZDiff*ZDiff);
    MoveX = (Route[PointNum].XPos - CharXPos) / Length * Speed;
    MoveZ = (Route[PointNum].ZPos - CharZPos) / Length * Speed;
}

```

### Следование за другим персонажем

Хотя сперва следование за другим персонажем может показаться сложным, не следует слишком беспокоиться. Помните, что решение предполагает сохранять вещи простыми. Следование за персонажем также просто как и ходьба по маршруту. Поскольку персонаж всегда знает свои собственные координаты и координаты персонажа, за которым он следует, вы можете использовать ранее виденные функции для перемещения одного персонажа по направлению к другому.

Единственное различие здесь в том, что вы можете захотеть, чтобы персонаж держался на определенном расстоянии от того персонажа, за которым следует, как показано на рис. 12.5.



**Рис. 12.5.** Один персонаж выбирает другого, чтобы тот следовал за ним. Если сопровождаемый персонаж ближе заданного расстояния, следующий персонаж стоит. Если расстояние больше, персонаж, который следует за другим, должен придвинуться поближе

Зная координаты каждого персонажа (персонажа, за которым будут следовать, и персонажа, который будет следовать) вы можете сконструировать единую функцию, которая будет определять, в каком направлении должен перемещаться «следующий» персонаж:

```
void CalculateFollowMovement(
    float CharXPos,          // Координаты персонажа
    float CharZPos,          // Координаты персонажа
    float WalkSpeed,         // Скорость ходьбы персонажа
    float FollowXPos,        // Координаты преследуемого
    float FollowZPos,        // персонажа
    float FollowDistance,    // Дистанция следования
    float *MoveX,            // Переменные для перемещения
    float *MoveZ)
{
    // Для быстрой проверки расстояния
    FollowDistance *= FollowDistance;

    // Получаем расстояние между персонажами
    float XDiff = (float)fabs(FollowXPos - CharXPos);
    float ZDiff = (float)fabs(FollowZPos - CharZPos);
    float Length = XDiff*XDiff + ZDiff*ZDiff;

    // Если расстояние между персонажами меньше допустимого,
    // просто стоим
    if(Length < FollowDistance) {
        *MoveX = *MoveZ = 0.0f;
        return;
    }

    // Вычисляем шаг перемещения
    // на основе скорости ходьбы персонажа
    Length = sqrt(Length);
    *MoveX = (CharXPos - FollowXPos) / Length * WalkSpeed;
    *MoveZ = (CharZPos - FollowZPos) / Length * WalkSpeed;
}
```

Каждый раз, когда вы хотите обновить данные персонажа, следующего за другим, вам достаточно передать в функцию требуемые данные и переместить персонаж, используя возвращенные переменные передвижения.

## **Избегание персонажа**

Избегание означает перемещение одного персонажа как можно дальше от другого. Если персонаж, которого избегают, находится ближе, чем минимальное расстояние, избегающий персонаж перемещается в противоположном направлении с помощью функции **CalculateEvadeMovement**:

```
void CalculateEvadeMovement(
    float CharXPos,          // Координаты персонажа
    float CharZPos,          // Координаты персонажа
    float WalkSpeed,         // Скорость ходьбы персонажа
    float FollowXPos,        // Координаты избегаемого персонажа
    float FollowZPos,        // персонажа
    float EvadeDistance,     // Дистанция избегания
    float *MoveX,            // Переменные для передвижения
    float *MoveZ)
```

```
{
    // Для быстрой проверки расстояния
    FollowDistance *= FollowDistance;

    // Получаем расстояние между персонажами
    float XDiff = (float)fabs(FollowXPos - CharXPos);
    float ZDiff = (float)fabs(FollowZPos - CharZPos);
    float Length = XDiff*XDiff + ZDiff*ZDiff;

    // Если расстояние между персонажами больше заданного,
    // остаемся стоять
    if(Length > EvadeDistance) {
        *MoveX = *MoveZ = 0.0f;
        return;
    }

    // Вычисляем шаг перемещения
    // на основе скорости ходьбы персонажа
    Length = sqrt(Length);
    *MoveX = -((CharXPos - FollowXPos) / Length * WalkSpeed);
    *MoveZ = -((CharZPos - FollowZPos) / Length * WalkSpeed);
}
```

## Автоматическое управление персонажами

Когда в игру вступают скрипты, вы захотите иногда управлять персонажем игрока, например, чтобы направить сюжет по определенному пути. Здесь привлекается автоматическое управление. Автоматическое управление персонажем означает, что игра решает какой метод управления использовать и куда переместить персонажа.

*Автоматическое управление* (automatic control) размещается в параметрах искусственного интеллекта. Для временного управления персонажем игрока вы можете использовать следующие шаги:

1. Смените тип персонажа с PC на NPC.
2. Смените параметры искусственного интеллекта PC (который теперь NPC) на следование по маршруту (или другой тип перемещения).
3. Выполняйте перемещение и продолжайте обновление до тех пор, пока не будет достигнута последняя маршрутная точка или пока вы не захотите прекратить использование автоматического управления.
4. Переключите тип персонажа обратно на PC.

## Общение между персонажами

Верно, начинается беседа! Взаимодействие персонажей является важной частью ролевой игры, но вы серьезно задумывались о том, как реализовать беседу в игре? К счастью, есть простые способы, сделать так, чтобы ваши персонажи разговаривали друг с другом, и чтобы не отклоняться от легкого пути, позвольте мне показать вам основные методы взаимодействия персонажей.

## Говорящие куклы

Простейший для использования метод общения — *разговаривающие куклы* (*talking dummy*). В каждой ролевой игре есть хотя бы один персонаж (говорящая кукла), который снова и снова повторяет одно и то же без малейшего участия интеллекта. Программировать говорящие куклы в вашей игре легко — назначьте строку текста, которая будет отображаться, когда с персонажем разговаривают.

Проблема в том, что говорящая кукла, произносящая всегда одну и ту же фразу, не слишком полезна. Также, вместо того, чтобы встраивать код для игровых диалогов в движок игры, вы можете использовать для общения внешние источники, что подводит нас к следующей теме, показывающей как усовершенствовать базовый проект говорящих кукол.

### Управляемые скриптами говорящие куклы

Вы знали, что это будет, не так ли? Скрипты — это сердце и душа компьютерных ролевых игр, так что вы должны пытаться использовать их в полной мере, в том числе и когда ваши персонажи общаются между собой. Путем назначения скрипта каждому персонажу вашей игры, скриптовый движок может взять базовую концепцию говорящих кукол и расширить ее.

Добавление возможности использовать в скриптах условный код позволяет говорящим куклам принимать решения о том, что сказать, основываясь на внутренних флагах и переменных. Предположим, у вас есть скрипт, который отслеживает состояние флага, указывающего посетили ли вы соседний город.

Когда в дело вступает управляемая скриптом говорящая кукла, ваш скриптовый движок определяет, какой текст показать, на основании полученного флага. Такой персонаж (кукла) посоветовал бы вам посетить соседний город, или, если вы уже были там, высказался бы о населении того города. Подобный скрипт может выглядеть так (в текстовом формате):

```
If flag 0 is TRUE then
    Print message "Я вижу, вы посетили Грэнуолл на юге!"
Else
    Print message "Вы должны сходить на юг в Грэнуолл."
Endif
```

Как видите, в показанном выше скрипте отслеживается равно ли значение флага (**flag 0**) **TRUE** или **FALSE** (флаг становится равным **TRUE**, всякий раз, когда игрок посещает город Грэнуолл).

Управляемые скриптами говорящие куклы относительно просты для создания и работы с ними, и я использую этот метод общения в оставшейся части книги. В идущем далее разделе «Демонстрация персонажей в программе Chars» и в главе 16 вы увидите обработку скриптов в действии и как использовать управляемые скриптами говорящие куклы в вашей игре.



## Показ диалогов и другого текста

Независимо от того, какой путь вы выберете, вам тем или иным способом надо будет отображать беседы между персонажами. Вы знаете процедуру — каждый раз, когда ваш игрок говорит с другим персонажем, выскакивает маленькое окошко, отображающее текст. Время от времени персонаж может выбирать из списка предоставленных действий и беседа продолжается.

Используя двухмерные техники вы можете показать окно беседы (или, если быть более точным, текстовое окно) с отображаемым внутри него текстом беседы. Поскольку персонаж может одновременно поместить в окно не очень много текста, отображается несколько окон с отдельными страницами, содержащими части полной беседы. Игрок нажимает кнопки для навигации по отображаемым в окне страницам текста. Когда текст завершается, беседа заканчивается.

Чтобы все было проще, я разработал систему (класс текстового окна, названный **cWindow**), которая может визуализировать текстовое окно любого размера в любом месте экрана. Окно может быть в любой момент перемещено и может содержать произвольную строку текста, которую вы назначите. В качестве дополнительного приза текстовое окно может действовать как вариант «текстового пузыря» с указателем на говорящий персонаж. На рис. 12.6 показан класс текстового окна в действии.



**Рис. 12.6.** Класс текстового окна **cWindow** позволяет вам открыть окно любой ширины и высоты для отображения любого текста (особенно текста беседы)

Технически окно представляет собой два прямоугольника, нарисованных один поверх другого, содержащиеся в буфере вершин. Один прямоугольник белый и чуть больше внутреннего цветного прямоугольника.

Когда вы рисуете их в правильном порядке (сначала большой белый прямоугольник, а затем меньший цветной прямоугольник), то получаете окно с рамкой, выглядящее как показано на рис. 12.6.

Текст в окне рисуется поверх этих двух прямоугольников. Текст может быть установлен в любое время, но предварительное задание строки текста дает дополнительную возможность вычисления размера окна, гарантирующего соответствие окна размеру строки. Когда вы определите размер окна, можно динамически менять рисуемую в нем строку текста без повторного создания определяющего окно буфера вершин.

В реальности вы можете использовать текстовое окно и для чего-нибудь другого. Скажем, вы можете открыть окно для показа изображения, используя класс текстового окна и объект текстуры. Это становится вопросом рисования сначала одного, а потом другого. В главе 16 вы увидите, как класс текстового окна находит применение для вещей, отличных от общения персонажей.

## Класс *cWindow*

Чтобы сдвинуться с мертвой точки, взглянем на приведенное ниже определение класса **cWindow**:

```
class cWindow
{
private:
    typedef struct sVertex { // Свои вершины
        float x, y, z;       // Координаты в экранном пространстве
        float rhw;           // Значение RHW
        D3DCOLOR Diffuse;    // Рассеиваемый цвет
    } sVertex;
#define WINDOWFVF (D3DFVF_XYZRHW | D3DFVF_DIFFUSE)
```

Помните, что текстовое окно использует буфер вершин для хранения пары прямоугольников (по два треугольника для определения каждого прямоугольника). Буфер вершин использует только преобразованные вершины, которым назначен рассеиваемый цвет (белый для большего прямоугольника на заднем плане и выбираемый вами цвет для меньшего прямоугольника поверх). Каждая вершина сохраняется в представленной (с соответствующим дескриптором вершины) структуре **sVertex**.

Затем вы определяете ряд указателей на родительские объекты **cGraphics** и **cFont**. У текстового окна для работы должны быть заранее инициализированный объект графического устройства и объект шрифта. Также вы объявляете экземпляр объекта буфера вершин текстового окна.

```
cGraphics *m_Graphics;    // Родительский объект cGraphics
cFont *m_Font;            // Объект шрифта
cVertexBuffer m_WindowVB; // Буфер вершин для окна
char *m_Text;             // Отображаемый текст
D3DCOLOR m_TextColor;    // Цвет для рисования текста
```

Текстовая строка содержится внутри класса (то есть в выделенном буфере **char**), вместе с соответствующим цветом, используемым для

рисования текста. После этого определения текстовой строки следуют координаты и размеры окна и отдельная переменная, указывающая надо ли рисовать указатель текстового пузыря (как определено последующим вызовом позиционирования окна).

```
long m_XPos, m_YPos;    // Координаты окна
long m_Width, m_Height; // Размеры окна
BOOL m_DrawTarget;     // Флаг для рисования указателя пузыря
```

Затем идут объявления открытых функций класса. (Я рассмотрю детали описания прототипов функций после их показа. С этого момента я буду показывать код каждой функции отдельно.)

```
public:
    cWindow(); // Конструктор
    ~cWindow(); // Деструктор

    // Функции создания/освобождения текстового окна
    BOOL Create(cGraphics *Graphics, cFont *Font);
    BOOL Free();

    // Устанавливаем текст и координаты/размеры/цвет окна
    BOOL SetText(char *Text, D3DCOLOR TextColor = 0xFFFFFFFF);

    // Перемещаем окно
    BOOL Move(long XPos, long YPos, long Width, long Height=0,
              long TargetX = -1, long TargetY = -1,
              D3DCOLOR BackColor = D3DCOLOR_RGBA(0, 64, 128, 255));
    long GetHeight(); // Получаем высоту окна после установки

    // Визуализация окна и отображение текста
    BOOL Render(char *Text = NULL);
};
```

### **cWindow::cWindow и cWindow::~cWindow**

Конструктор и деструктор малы и по сути только очищают и освобождают ресурсы класса:

```
cWindow::cWindow()
{
    // Очистка данных класса
    m_Graphics = NULL;
    m_Font = NULL;
    m_Text = NULL;
    m_TextColor = 0xFFFFFFFF;
}

cWindow::~cWindow()
{
    Free(); // Освобождение данных класса
}
```

## cWindow::Create и cWindow::Free

Вы используете функции **Create** и **Free** для подготовки класса к использованию (путем назначения объектов **cGraphics** и **cFont**, применяемых для визуализации) и освобождения внутренних данных класса:

```

BOOL cWindow::Create(cGraphics *Graphics, cFont *Font)
{
    Free(); // Освобождаем предыдущие данные класса

    // Проверка ошибок
    if((m_Graphics = Graphics) == NULL || (m_Font = Font) == NULL)
        return FALSE;

    // Создаем новый буфер вершин
    // (с 11 вершинами для использования)
    m_WindowVB.Create(m_Graphics, 11, WINDOWFVF, sizeof(sVertex));

    return TRUE; // Возвращаем флаг успеха
}

BOOL cWindow::Free()
{
    m_WindowVB.Free(); // Освобождаем буфер вершин
    delete [] m_Text; // Удаляем текстовый буфер
    m_Text = NULL;
    return TRUE;
}

```

Буфер вершин окна использует 11 вершин для хранения двух прямоугольников окна и графического указателя на местоположение цели на экране. Вы увидите, как этот буфер создается и используется немного позже в разделе «cWindow::Move». Что касается текста окна, достаточно только выделить массив и скопировать текст в него, так что освобождение класса включает удаление используемого массива.

## cWindow::SetText

Как я только что упомянул, вы храните текст окна, создавая массив байтов и копируя текст окна в этот массив. Установка текста окна — это предназначение функции **SetText**, которая получает два параметра — используемый текст (**char \*Text**) и цвет, используемый для рисования текста (**D3DCOLOR TextColor**).

```

BOOL cWindow::SetText(char *Text, D3DCOLOR TextColor)
{
    // Удаление предыдущего текста
    delete [] m_Text;
    m_Text = NULL;

    m_Text = strdup(Text); // Копирование текстовой строки
    m_TextColor = TextColor; // Сохранение цвета текста
    return TRUE;
}

```

Для эффективности используется функция **strdup**, которая сразу выделяет память и копирует текстовую строку. Функция **strdup** получает в своем аргументе текстовую строку и возвращает указатель на выделенный буфер, который содержит текст из запроса (вместе с завершающим символом **NULL**, который отмечает конец текста). Теперь текст готов к использованию в классе, и вы в любой момент можете поменять текст, просто вызвав **SetText** еще раз.

### **cWindow::Move**

Самая большая в связке функция, **cWindow::Move**, выполняет работу по конструированию буфера вершин, используемого для визуализации окна (и вспомогательного указателя, если надо). Функция получает в аргументах местоположение окна (экранные координаты), размеры (в пикселях), пару координат для точки из которой будет исходить указатель текстового пузыря, и цвет, используемый для внутренней области окна.

```
BOOL cWindow::Move(long XPos, long YPos,
                  long Width, long Height,
                  long TargetX, long TargetY,
                  D3DCOLOR BackColor)
{
    sVertex Verts[11];
    long i;
```

После объявления нескольких используемых локальных переменных, вы сохраняете местоположение и размеры окна в переменных класса. Прототип функции **Move** по умолчанию присваивает аргументу **Height** значение 0; это значит, что вы позволяете классу вычислить высоту, требуемую для отображения текста, содержащегося в уже созданном текстовом буфере.

Замечательно, что функция **Move** вычисляет высоту текста. Если вы отображаете текст неизвестной длины, достаточно присвоить **Height** значение 0 и пусть класс выполняет всю сложную работу. Если говорить об этой тяжелой работе, код, сохраняющий местоположение и размеры таков (включая код для вычисления высоты):

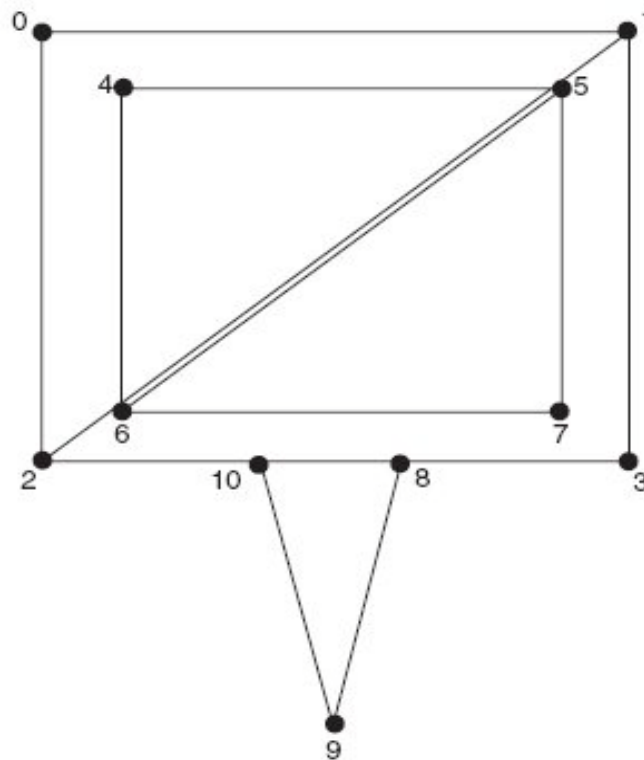
```
// Сохраняем координаты и вычисляем если надо высоту
m_XPos = XPos;
m_YPos = YPos;
m_Width = Width;

if(!(m_Height = Height)) {
    RECT Rect;
    Rect.left = XPos;
    Rect.top = 0;
    Rect.right = XPos + Width - 12;
    Rect.bottom = 1;

    m_Height = m_Font->GetFontCOM()->DrawText(m_Text, -1,
                                              &Rect, DT_CALCRECT | DT_WORDBREAK,
                                              0xFFFFFFFF) + 12;
}
```

Далее код посвящен конструированию буфера вершин для отображения окна. Как я упоминал, вы используете два прямоугольника, и каждый из них использует четыре вершины (организованные в полосу треугольников для экономии памяти и улучшения времени визуализации).

Используя ранее сохраненные местоположение и размеры, окно конструируется, как показано на рис. 12.7.



**Рис. 12.7.** Два окна, каждое из которых использует четыре вершины и определено с использованием полосы треугольников. Последний треугольник в буфере вершин использует собственные три вершины и хранится как список треугольников

Кроме того, буфер вершин может содержать еще три вершины, которые образуют указатель в верхней или в нижней части текстового окна. Присваивание аргументам **TargetX** и **TargetY** значений, отличных от  $-1$ , информирует функцию **Move** о необходимости рисовать указатель. Вот код для установки вершин окна:

```
// Очищаем данные вершин
for(i = 0; i < 11; i++) {
    Verts[i].z = 0.0f;
    Verts[i].rhw = 1.0f;
    Verts[i].Diffuse = 0xFFFFFFFF;
}

// Устанавливаем белый бордюр
Verts[0].x = (float)m_XPos;
Verts[0].y = (float)m_YPos;
Verts[1].x = (float)(m_XPos+m_Width);
Verts[1].y = (float)m_YPos;
Verts[2].x = (float)m_XPos;
Verts[2].y = (float)(m_YPos+m_Height);
Verts[3].x = (float)(m_XPos+m_Width);
Verts[3].y = (float)(m_YPos+m_Height);
```

```
// Устанавливаем текстовое окно
Verts[4].x = (float)m_XPos+2.0f;
Verts[4].y = (float)m_YPos+2.0f;
Verts[4].Diffuse = BackColor;
Verts[5].x = (float)(m_XPos+m_Width)-2.0f;
Verts[5].y = (float)m_YPos+2.0f;
Verts[5].Diffuse = BackColor;
Verts[6].x = (float)m_XPos+2.0f;
Verts[6].y = (float)(m_YPos+m_Height)-2.0f;
Verts[6].Diffuse = BackColor;
Verts[7].x = (float)(m_XPos+m_Width)-2.0f;
Verts[7].y = (float)(m_YPos+m_Height)-2.0f;
Verts[7].Diffuse = BackColor;

// Устанавливаем указатель цели
if(TargetX != -1 && TargetY != -1) {
    m_DrawTarget = TRUE;

    if(TargetY < m_YPos) {
        Verts[8].x = (float)TargetX;
        Verts[8].y = (float)TargetY;
        Verts[9].x = (float)(m_XPos+m_Width/2+10);
        Verts[9].y = (float)m_YPos;
        Verts[10].x = (float)(m_XPos+m_Width/2-10);
        Verts[10].y = (float)m_YPos;
    } else {
        Verts[8].x = (float)(m_XPos+m_Width/2-10);
        Verts[8].y = (float)(m_YPos+m_Height);
        Verts[9].x = (float)(m_XPos+m_Width/2+10);
        Verts[9].y = (float)(m_YPos+m_Height);
        Verts[10].x = (float)TargetX;
        Verts[10].y = (float)TargetY;
    }
} else {
    m_DrawTarget = FALSE;
}

m_WindowVB.Set(0, 11, &Verts); // Устанавливаем вершины
return TRUE;
}
```

### cWindow::GetHeight

Поскольку вы не можете задать значение высоты в вызове **Move**, вам может потребоваться получить действительно используемую высоту текстового окна. **GetHeight** делает именно это; она возвращает высоту текстового окна, которая была определена в **Move**:

```
long cWindow::GetHeight()
{
    return m_Height;
}
```

### cWindow::Render

Последней выступает функция **Render**, которую вы вызываете в кодовом блоке между **cGraphics::BeginScene** и **cGraphics::EndScene**.

**Render** просто устанавливает требуемые режимы визуализации и рисует необходимые полигоны, образующие указатель и текстовое окно. Затем рисуется текстовая строка (если высота окна больше 12, что является размером рамки, используемой для обрамления меньшей внутренней части).

```

BOOL cWindow::Render(char *Text, D3DCOLOR Color)
{
    // Проверка ошибок
    if(m_Graphics == NULL || m_Font == NULL)
        return FALSE;

    // Установка режимов визуализации
    m_Graphics->SetTexture(0, NULL);
    m_Graphics->EnableZBuffer(FALSE);

    // Рисуем окно
    m_WindowVB.Render(0, 2, D3DPT_TRIANGLESTRIP);
    m_WindowVB.Render(4, 2, D3DPT_TRIANGLESTRIP);

    // Рисуем указатель, если надо
    if(m_DrawTarget)
        m_WindowVB.Render(8, 1, D3DPT_TRIANGLELIST);

    // Рисуем текст, только если высота > 12
    if(m_Height > 12) {
        // Рисуем текст
        if(Text == NULL)
            m_Font->Print(m_Text, m_XPos+6, m_YPos+6,
                          m_Width-12, m_Height-12,
                          m_TextColor, DT_WORDBREAK);
        else
            m_Font->Print(Text, m_XPos+6, m_YPos+6,
                          m_Width-12, m_Height-12,
                          Color, DT_WORDBREAK);
    }
    return TRUE;
}

```

У **Render** есть два необязательных аргумента. Первый аргумент, **Text**, позволяет переопределить текст, заданный внутри класса, который был установлен с использованием функции **SetText**. Переопределение рисуемого текста замечательно подходит для динамического обновления того, что нужно показать. Второй аргумент, **Color**, задает цвет, который вы хотите использовать для рисования отображаемого текста.

## Использование cWindow

Чтобы быстро продемонстрировать использование **cWindow**, позвольте мне показать как отобразить короткое сообщение (подразумевается, что вы уже инициализировали объект **cGraphics**):

```

// Graphics = ранее инициализированный объект cGraphics
cFont Font;
cWindow Window;

// Создание используемого шрифта
Font.Create(&Graphics, "Arial", 16);

```



```
// Подготавливаем класс для использования
Window.Create(&Graphics, &Font);
Window.SetText("The cWindow class in action!");
Window.Move(4, 4, 632);

Graphics.BeginScene(); // Начинаем сцену
Window.Render();        // Рисуем окно и текст
Graphics.EndScene();    // Завершаем сцену
Graphics.Display();     // Отображаем сцену

Window.Free(); // Освобождаем данные класса окна
```

---

### ВНИМАНИЕ!

Сейчас вы должны жестко задавать размер шрифта; шрифт должен быть 16 пикселей, иначе могут возникнуть проблемы с межстрочными интервалами. Я оставляю изменение кода для вас, но во всех последующих примерах я настоятельно рекомендую придерживаться использования шрифта Arial размером 16.

---

Полезность класса **cWindow** будет полностью видна позже, в разделе «Демонстрация персонажей в программе Chars», когда вы будете отображать диалоги, так что пока отложите общение и текстовые окна на второй план.

## Скрипты и персонажи

Скрипты в этих нескольких последних главах повсюду высовывают свою голову, и вырисовывается истина: скрипты играют главную роль, когда имеешь дело с персонажами. Скрипты работают с общением, заклинаниями, перемещением персонажей и многим другим.

Что вам надо сейчас, так это четко определить метод обработки игровых скриптов. Лучший способ сделать это состоит в создании класса сплетающего воедино вашего персонажа и другую прикладную обработку.

### Класс скрипта

Вместо того, чтобы разрабатывать класс обработки скриптов раньше в этой книге, я подождал, пока в нем не возникнет настоящая необходимость. В главе 10, «Реализация скриптов», вы узнали как легко работать с системой Mad Lib Script (MLS) и как просто обрабатывать скрипты, созданные с использованием MLS Editor. Однако, выполнение скриптов становится еще проще, когда вы помещаете обработку скриптов целиком в класс. Здесь вам будет представлен класс **cScript**, определенный в файлах Script.cpp и Script.h (вы найдете их в каталоге \BookCode\Chap12\Chars на CD).

```
class cScript
{
private:
    long m_NumActions; // Количество загруженных действий скрипта
    sScript *m_ScriptParent; // Связанный список скрипта
```

```

// Перегружаемые функции для подготовки к обработке скрипта
// и когда обработка завершена
virtual BOOL Prepare() { return TRUE; }
virtual BOOL Release() { return TRUE; }

// Обработка отдельного действия скрипта
virtual sScript *Process(sScript *Script)
    { return Script->Next; }

public:
    cScript(); // Конструктор
    ~cScript(); // Деструктор

    BOOL Load(char *Filename); // Загрузка скрипта
    BOOL Free(); // Освобождение загруженного скрипта
    BOOL Execute(char *Filename=NULL); // Выполнение скрипта
};

```

Обманчиво маленький класс **cScript** обладает недюжинной мощностью. Загрузка скрипта выполняется через функцию **Load**. После загрузки вы можете выполнить скрипт с помощью вызова **Execute**. Если вы не хотите возиться с загрузкой скрипта перед его выполнением, вызов функции **Execute** с указанием файла скрипта загрузит его и выполнит в одном вызове функции (плюс освободит скрипт по завершении выполнения).

---

<b>ПРИМЕЧАНИЕ</b>	Использование функции <b>Load</b> для загрузки скрипта полезно, когда скрипт выполняется несколько раз, поскольку вы не будете освобождать его между использованиями. Загрузка скрипта через функцию <b>Execute</b> вынуждает каждый раз загружать и освобождать его, что приводит к напрасным тратам времени.
-------------------	--

---

Способ обработки скрипта классом **cScript** весьма оригинален. В действительности вы должны выполнить наследование от класса **cScript** для разбора и выполнения каждого действия скрипта. Это цель функции **Process**. Когда скрипт загружен, функция **Process** вызывается для обработки каждого действия скрипта.

Через указатель на скрипт запрашивается номер действия скрипта и вы должны решить, что делать с этим действием. Затем вам надо обновить указатель на скрипт, вернув указатель на следующее действие скрипта в связанном списке. (Обратитесь к главе 10, за информацией о том, как обрабатываются скрипты.)

Полный код класса **cScript** был показан в главе 10, так что давайте теперь повернемся к рассмотрению того, как выполнить наследование класса, используемого для обработки скрипта.

## Создание производного класса скрипта

Я предполагаю, что к этому моменту вы можете свободно работать с шаблонами действий и скриптами. В качестве примера использования

производного класса скрипта будет рассмотрен следующий шаблон действий:

```
"End"
"If flag ~ equals ~ then"
    INT 0 255
    BOOL
"Else"
"EndIf"
"Set flag ~ to ~"
    INT 0 255
    BOOL
"Print ~"
    TEXT
```

Теперь, используя представленный шаблон действий, напишем следующий скрипт (я привожу его в текстовой форме, чтобы он был проще для понимания):

```
If flag 0 equals TRUE then
    Print "Flag is TRUE"
    Set flag 0 to FALSE
Else
    Print "Flag is FALSE"
    Set flag 0 to TRUE
EndIf
```

---

**ВНИМАНИЕ!**

Помните, что пример скрипта показан здесь в текстовой форме, но когда используется Mad Lib Script формат основан на значениях. Например, действие **If...then** представлено значением 1, а действие **Endif** — значением 3.

---

Краткий просмотр показывает, что представленный скрипт отображает сначала сообщение «Flag is FALSE» (поскольку всем флагам скрипта при инициализации присваивается **FALSE**); когда он выполняется снова, скрипт отображает «Flag is TRUE».

### ***Производный класс***

Следующий шаг для обработки скрипта — наследование класса от **cScript**:

```
class cGameScript : public cScript
{
    private:
        BOOL m_Flags[256]; // Внутренние флаги

        // Прототипы функций скрипта
        sScript *Script_End(sScript*);
        sScript *Script_IfFlagThen(sScript*);
        sScript *Script_Else(sScript*);
        sScript *Script_EndIf(sScript*);
        sScript *Script_SetFlag(sScript*);
        sScript *Script_Print(sScript*);
```

```

// Перегруженная функция обработки
sScript *Process(sScript *Script);

public:
    cGameScript();
};

```

Показанный здесь производный класс (**cGameScript**) использует массив значений **BOOL**, представляющий внутренние флаги, которые скрипт может использовать. Следом за единственным объявлением переменной идет список прототипов функций.

Прототипы функций скрипта — это средства к существованию обработчика скрипта. У каждого действия скрипта есть назначенная функция, которая вызывается из функции **Process**. Как вы увидите чуть позже в этом разделе, функция **Process** переопределяется, чтобы вызывать именно эти функции скрипта.

Помимо этих закрытых функций есть конструктор, который очищает массив **m\_Flags**, присваивая всем значениям **FALSE**:

```

cGameScript::cGameScript()
{
    // Сброс всех внутренних флагов в FALSE
    for(short i = 0; i < 256; i++)
        m_Flags[i] = FALSE;
}

```

Вернемся немного назад и взглянем на переопределенную функцию **Process**. Как видно из последующего кода, **cGameScript::Process** получает тип текущего действия скрипта и передает управление соответствующей функции. При возврате из каждой функции действия возвращается указатель на следующее действие скрипта. Если возвращается значение **NULL**, выполнение скрипта останавливается.

```

sScript *cGameScript::Process(sScript *Script)
{
    // Переход к функции на основании типа действия
    switch(Script->Type) {
        case 0: return Script_End(Script);
        case 1: return Script_IfFlagThen(Script);
        case 2: return Script_Else(Script);
        case 3: return Script_EndIf(Script);
        case 4: return Script_SetFlag(Script);
        case 5: return Script_Print(Script);
    }
    return NULL; // Ошибка выполнения
}

```

Теперь, когда вы переопределили функцию **Process** (и заполнили инструкцию **switch** вызовами функций действий), можно продолжить и запрограммировать все функции действий, как показано ниже:

```
sScript *cGameScript::Script_End(sScript *Script)
{
    // Принудительная остановка обработки скрипта
    return NULL;
}

sScript *cGameScript::Script_IfFlagThen(sScript *Script)
{
    BOOL Skipping; // Флаг для условия if...then

    // Смотрим, соответствует ли флаг второму элементу
    if(m_Flags[Script->Entries[0].lValue % 256] ==
        Script->Entries[1].bValue)
        Skipping = FALSE; // Не пропускаем следующие действия
    else
        Skipping = TRUE;  // Пропускаем следующие действия

    // Сейчас Skipping установлена, если действия скрипта
    // надо пропустить, согласно условной инструкции if...then.
    // Дальнейшие действия обрабатываются, если skipped = FALSE,
    // при этом смотрим не появится ли else для переключения
    // режима пропуска или endif для завершения условного блока
    Script = Script->Next; // Переходим к обработке следующего действия

    while(Script != NULL) {
        // Если это else, переключаем режим пропуска
        if(Script->Type == 2)
            Skipping = (Skipping == TRUE) ? FALSE : TRUE;

        // Прерывание в конце if
        if(Script->Type == 3)
            return Script->Next;

        // Обрабатываем функции скрипта в условном блоке
        // обеспечивая пропуск действий, когда условие не выполняется
        if(Skippping == TRUE)
            Script = Script->Next;
        else {
            if((Script = Process(Script)) == NULL)
                return NULL;
        }
    }
    return NULL; // Достигнут конец скрипта
}

sScript *cGameScript::Script_Else(sScript *Script)
{
    return Script->Next; // Переходим к следующему действию скрипта
}

sScript *cGameScript::Script_EndIf(sScript *Script)
{
    return Script->Next; // Переходим к следующему действию скрипта
}

sScript *cGameScript::Script_SetFlag(sScript *Script)
{
    // Установка логического значения
    m_Flags[Script->Entries[0].lValue % 256] =
        Script->Entries[1].bValue;
    return Script->Next; // Переходим к следующему действию скрипта
}
```

```
sScript *cGameScript::Script_Print(sScript *Script)
{
    // Отображаем текст в окне сообщений
    MessageBox(NULL, Script->Entries[0].Text, "Text", MB_OK);
    return Script->Next; // Переходим к следующему действию скрипта
}
```

## Использование производного класса

Чтобы проверить класс **cGameScript**, создайте его экземпляр и запустите пример скрипта, показанный мной ранее в разделе «Создание производного класса скрипта». Если предположить, что вы сохранили скрипт в файл с именем `test.mls`, функциональность класса скрипта вам покажет следующий пример:

```
cGameScript Script;

Script.Execute("test.mls"); // Печатает сообщение Flag is FALSE
// Теперь внутренние флаги скрипта поддерживаются,
// так что следующий вызов учитывает новое состояние флага
Script.Execute("test.mls"); // Печатает сообщение Flag is TRUE
```

Хотя этот пример производного класса **cGameScript** сляпан на быструю руку, он в действительности не слишком отличается от законченного анализатора скриптов, использующего огромный шаблон действий. Вам просто надо добавить в класс каждую функцию обработки действия, и вызывать их через функцию **Process**. Повсюду в оставшейся части книги вы будете видеть применение этого класса скрипта; фактически, он формирует основу многих проектов.

## Управление ресурсами

Управление ресурсами играет одну из основных ролей в вашей игре. Чтобы добраться куда-нибудь или достигнуть чего-либо персонажи, возможно, будут нуждаться в помощи предметов, заклинаний и других объектов, сконструированных вами для игры.

В главе 11, «Определение и использование объектов», вы увидели, как разработать предметы вашей игры и поместить их в главный список предметов. теперь пришло время посмотреть, как вы можете обращаться с этими предметами.

Ресурсы также включают заклинания, которые ваш игровой персонаж может выучить по ходу игры. О заклинаниях вы больше прочитаете чуть дальше в этой главе, в разделе «Работа с магией и заклинаниями», а сейчас вот краткий обзор того, что ресурсы могут сделать для вас.

## Использование предметов

Вы уже видели, как управлять предметами в вашей игре, используя систему управления имуществом персонажа. Если вы экспериментировали с

программой CharICs, то заметили отсутствие функций для кнопок **Equip** и **Use**. Теперь пришло время понять, что случится, когда эти функциональные возможности будут помещены на свое место в том примере.

Персонаж должен отслеживать, какой броней, оружием и аксессуарами он экипирован. Поскольку эти предметы экипировки хранятся в структуре **sItem**, вы можете запросить связанное с ними значение модификатора (хранящееся как **sItem::Value**), используемое для изменения способностей персонажа.

Скажем, предмет 1 это меч. Этот меч определен как оружие и имеет значение модификатора 10. Если персонажу разрешено экипироваться оружием (это определяется через флаги класса), соответствующее значение модификатора прибавляется к значению способности атаковать персонажа. Чтобы отслеживать, какие предметы есть в экипировке, достаточно сохранить номер предмета в описании персонажа. Также, поскольку вы хотите визуально показать, какое оружие держит персонаж, вы можете загрузить сетку оружия для присоединения ее к персонажу.

Что касается использования предметов, оно настолько же просто, как и экипировка. Вспомните, что вы заранее определили, чем может быть каждый предмет и что он может сделать. Предположим, у вас есть целебный эликсир, и игрок решает выпить его. Проверив ограничения использования, вы определите, что целебный эликсир является лечащим предметом и, следовательно, добавляет значение модификатора предмета к очкам здоровья игрока.

## Использование магии

Магические заклинания, также как и предметы, являются основополагающим ресурсом для выживания игрока. Магия помогает пользователю различными путями; усиление атаки, улучшение защиты и исцеление — типичные результаты произнесения магических заклинаний. Хотя заклинания не являются абсолютно необходимыми, будьте уверены — они действительно помогают.

Магические заклинания определяются точно так же, как предметы. Они имеют назначенные возможности. Одни заклинания меняют здоровье персонажей, другие заклинания могут вызывать изменение дополнительных статусов (таких, как отравление). Любой персонаж может использовать волшебство; он просто должен «знать» заклинание и иметь достаточно очков маны для его произнесения.

Чтобы персонаж знал заклинание, он должен изучить его, работая над своим уровнем опыта. Вы можете отслеживать выученные заклинания, используя битовые флаги. Можно использовать для хранения известных заклинаний 32-разрядную переменную, где каждый бит представляет конкретное заклинание. Если разряд в переменной установлен, заклинание выучено. Выученное заклинание обычно никогда не забывается.

Управление использованием заклинаний аналогично управлению предметами; отображается список известных персонажу заклинаний, где игрок может выбрать, какое заклинание произнести, когда и где. В этой книге нет никаких ограничений когда и где персонаж может произносить заклинания.

В разделе «Работа с магией и заклинаниями» вы больше узнаете о включении заклинаний в ваш проект.

## Торговля и обмен

Ресурсы являются товаром, и игроки будут хотеть покупать и продавать свое имущество друг другу. Только определенные персонажи в игре открыты для обмена. Эти персонажи обычно называются «лавочники», поскольку имеют тенденцию появляться только на стоянках. Вы знаете процедуру — заходим в магазин, подходим к прилавку и начинаем сделку.

Есть отдельные типы магазинов для работы с каждым типом ресурсов — продуктовые лавки, оружейные магазины, лавки с доспехами и т.д. Вы можете использовать стандартный подход к магазинам во всех их проявлениях.

Система управления имуществом персонажа подходит не только для персонажа игрока; система управления имуществом (ICS), разработанная в главе 11, замечательно работает с магазинами и лавочниками. У персонажа лавочника есть присоединенная к нему уникальная ICS, которая определяет, какие предметы данный персонаж лавочника может продать. Не надо беспокоиться о покупке предметов лавочниками; все лавочники могут приобретать все предметы, отмеченные как продаваемые (естественно, занижая цену, определенную в описании предмета).

Покупка предметов у лавочника заключается в отображении списка предметов лавочника и их стоимости. Обычно запасы предметов на складе лавочника никогда не иссякают, независимо от того, сколько предметов купит игрок; но иногда вы будете хотеть, чтобы лавочник продавал только один экземпляр предмета.

Немного изменим ICS, но только в понятии термина количества предметов. Если у лавочника неограниченное количество какого-либо предмета, установим количество этого предмета равным 2 или более (обратитесь к главе 11, чтобы посмотреть, как установить количество предметов). Значение количества равное 1 означает, что лавочник может продать данный предмет только один раз. Изобретательно, не так ли?

Вы можете обнаружить, что работать с магазинами лучше вне кода персонажа, в основном коде игрового приложения. Посмотрите главу 16, чтобы увидеть, как я реализовал систему товарооборота.



## Работа с магией и заклинаниями

Естественно, в первоклассной ролевой игре должны быть персонажи, способные запрячь таинственные магические силы, чтобы превратить жителей игры в небольшие комочки обугленной плоти. Даже если вы не на смертельной стороне магии, не отклоняйте пользу своевременного целебного заклинания. Волшебство очень важно в ролевых играх, и пришло время посмотреть как экипировать ваших игроков удивительными заклинаниями, оказывающими мощный эффект на их цели.

С точки зрения игрока, заклинание это восхитительная вспышка графических эффектов, тогда как со стороны игрового движка это всего лишь функция, меняющая данные персонажей. Ваш игровой движок может разделить графику и функциональность, два компонента заклинания, чтобы поддержка каждого из компонентов была проще.

### Графика заклинания

Используя трехмерные сетки, вы можете легко иметь дело с графической частью заклинаний. Эти заклинания начинаются от волшебника, движутся вперед к их намеченным целям, и в конечной точке производят сокрушительное действие на какого-нибудь несчастного персонажа. Это происходит в три этапа — возникновение сетки заклинания, перемещение сетки и когда сетка достигает цели. На каждом этапе вы можете назначить анимированную сетку, а это значит, что у каждого заклинания может быть до трех графически представляющих его сеток.

---

**ПРИМЕЧАНИЕ**

Каждая сетка отображается отдельно. Две сетки заклинания никогда не отображаются одновременно. Когда одна сетка завершает цикл, она освобождается, и новая сетка занимает ее место.

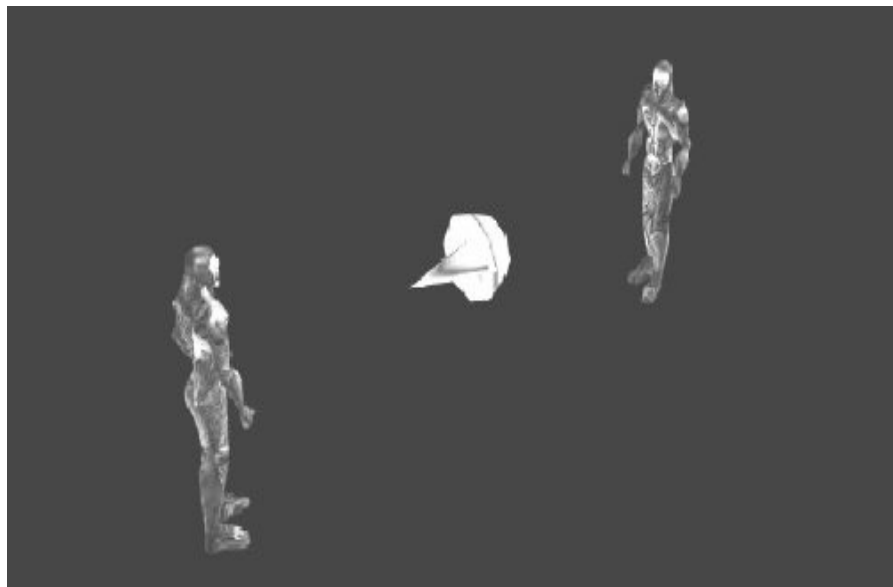
---

Чтобы увеличить количество возможных вариантов при создании эффектов заклинаний в вашей игре, местоположение и перемещение этих сеток не фиксировано. Фактически, любая из трех сеток может парить над заклинателем или целью, перемещаться от заклинателя к цели или от цели к заклинателю, либо растягиваться между заклинателем и целью.

Всякий раз, когда сетка парит около заклинателя или цели (или растянута между ними), это занимает фиксированный промежуток времени (измеряемый в миллисекундах). Это позволяет сетке завершить ее цикл анимации (или несколько циклов).

Что касается перемещения сеток (от заклинателя к цели или наоборот), сетке назначается скорость перемещения (измеряемая в единицах за секунду). Как только сетка достигнет цели, она удаляется и ее место занимает следующая сетка (если какая-нибудь сетка должна следовать за ней).

Предположим, у вас есть заклинание шаровой молнии. Для него требуется только две сетки. Первая сетка, шаровая молния, появляется у заклинателя и перемещается к цели, как показано на рис. 12.8.



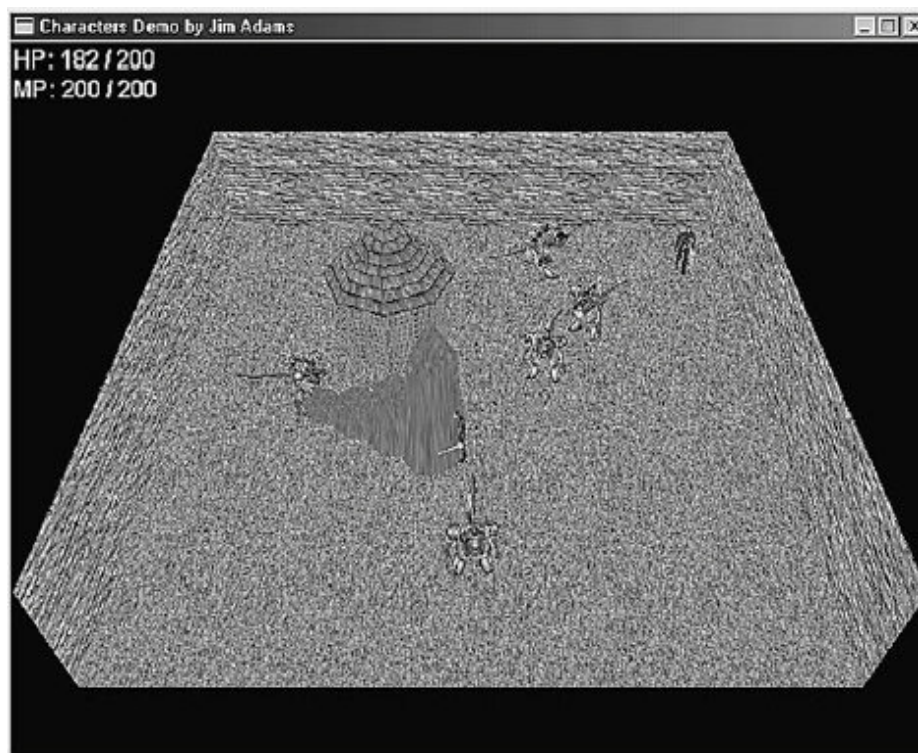
*Рис. 12.8. Заклинатель выпустил шаровую молнию в намеченную цель. Сетка перемещается с установленной скоростью, пока не достигнет места назначения*

Вторая сетка, взрыв, отображается когда первая сетка достигнет цели. Сетка взрыва располагается над целью и проходит несколько циклов анимации, чтобы создать впечатление действительно наносимого ущерба. Вы, возможно, задумываетесь о тех случаях, когда не хотите, чтобы заклинание перемещалось, и в то же время хотите, чтобы оно распространялось по направлению к вашей цели. Это причина для эластичного позиционирования сеток.

Если вы определяете сетку заклинания, распространяющуюся в положительном направлении оси Z (а вы так должны делать всегда), эта сетка может располагаться возле заклинателя и масштабироваться так, чтобы дальний край сетки касался цели. Подобное растягивание (или масштабирование) замечательно подходит для таких заклинаний, как молния или земляной вал (используемого в демонстрационной программе Chars из этой главы), которое взрывает землю между заклинателем и жертвой (как показано на рис. 12.9).

Демонстрационную программу Chars вы найдете на прилагаемом CD-ROM в каталоге \BookCode\Chap12\Chars. Хотя и незаметное на первый взгляд, заклинание земляного вала, описанное на рис. 12.9, показывает небольшую коричневую сетку, растянутую от заклинателя к целевому персонажу.

Как видите, отслеживание трех сеток графической составляющей заклинания, это вопрос загрузки соответствующей сетки и ее визуализации в правильном местоположении на заданный период времени. После завершения требуемого количества циклов анимации сетки графическая часть заклинания завершена, и пришло время перейти к функциональной части.



*Рис. 12.9. Заклинание земляного вала из демонстрационной программы Chars демонстрирует технику растягивающегося/масштабируемого позиционирования сетки. Независимо от расстояния между заклинателем и целью, сетка масштабируется так, чтобы всегда начинаться возле заклинателя и касаться цели*

## Функция заклинания

Функциональный компонент заклинания выполняет настоящую работу. Как только заклинание прошло через анимацию и достигло намеченной цели, необходимо иметь дело с его вредоносным или благоприятным воздействием.

Исцеляющее заклинание должно восстановить здоровье травмированного персонажа, в то время, как повреждающее заклинание должно лишать здоровья. Фактически, почти любой параметр персонажа может быть изменен заклинанием, начиная от здоровья и маны персонажа, и кончая его способностями, такими как атака или защита. В таблице 12.5 я описываю различные эффекты действия заклинаний, используемые в этой книге.

**Таблица 12.5.** Возможные эффекты заклинаний

Эффект	Описание
Изменение здоровья	Увеличивает или уменьшает на заданное количество очки здоровья цели.
Изменение маны	Увеличивает или уменьшает на заданное количество очки маны цели.
Снятие статуса	Снимает с цели конкретный дополнительный статус (такой, как отравление или замедление).

Таблица 12.5. Возможные эффекты заклинаний (продолжение)

Эффект	Описание
Установка статуса	Устанавливает дополнительный статус цели (отравление и т.д.).
Воскрешение мертвых	Возвращает РС обратно к жизни.
Мгновенная смерть	Немедленно убивает целевой персонаж.
Расколдовывание	Снимает все дополнительные статусы целевого персонажа.
Телепортация	Телепортирует РС в заданное местоположение на карте.

У каждого эффекта заклинания есть соответствующее значение, определенное в исходном коде как объект **enum**, описанный так:

```
enum SpellEffects {
    ALTER_HEALTH = 0,
    ALTER_MANA,
    CURE_AILMENT,
    CAUSE_AILMENT,
    RAISE_DEAD,
    INSTANT_KILL,
    DISPEL_MAGIC,
    TELEPORT
};
```

Каждому эффекту заклинания назначено число. Благодаря этому вы можете организовать обработку эффектов заклинаний в одной инструкции **switch**, например, так:

```
switch(SpellEffect) {
    case ALTER_HEALTH:
        // Обработка изменения здоровья

    case ALTER_MANA:
        // Обработка изменения маны
    ...
}
```

Каждый эффект заклинания достаточно прямолинеен. Теперь давайте пристальнее взглянем, что каждый из этих эффектов делает.

### Изменение здоровья и маны

Заклинание может нанести ущерб или отнять у персонажа накопленную им ману, или, с другой стороны, восстановить здоровье или ману. Изменение здоровья это, вероятно, наиболее широко используемый эффект заклинаний.

У заклинаний изменения здоровья есть связанное значение, определяющее, сколько здоровья отнимается или добавляется целевому персонажу. То же самое применимо к очкам маны, которые могут быть восстановлены или опустошены легким движением руки волшебника.

## **Установка и снятие статусов**

Дополнительные статусы — это яд и награда способностей и атрибутов персонажа. Дуэт из эффектов установки и снятия дополнительных статусов дает вам огромную свободу в выборе того, как проклясть или наградить персонаж.

Когда вы меняете (устанавливаете или снимаете) дополнительный статус, можно использовать для определения статусов битовое представление. Используя битовые флаги можно сразу установить или снять сразу несколько статусов. У каждого статуса есть связанное макроопределение, выглядящее так (все они находятся в файле `Chars.h`):

```
#define AILMENT_POISON          1
#define AILMENT_SLEEP          2
#define AILMENT_PARALYZE       4
#define AILMENT_WEAK           8
#define AILMENT_STRONG         16
#define AILMENT_ENCHANTED      32
#define AILMENT_BARRIER       64
#define AILMENT_DUMBFOUNDED    128
#define AILMENT_CLUMSY         256
#define AILMENT_SUREFOOTED     512
#define AILMENT_SLOW           1024
#define AILMENT_FAST           2048
#define AILMENT_BLIND          4096
#define AILMENT_HAWKEYE        8192
#define AILMENT_SILENCED       16384
```

Статусы персонажа хранятся в одной 32-разрядной переменной, и у всех персонажей есть связанная с ними переменная статусов. Верно, любой персонаж в вашей игре может быть обременен дополнительными статусами, но труднее сокрушить те персонажи, у которых высокая сопротивляемость.

## **Воскрешение и мгновенная смерть**

Время от времени ваши бедные персонажи будут умирать, и вы хотите, чтобы была возможность поднимать их из мертвых, за исключением врагов, конечно. Заклинание с эффектом воскрешения делает именно это — воскрешает РС или NPC из мертвых и дает ему одно очко здоровья.

С другой стороны, будут моменты, когда вы захотите свалить монстра одним ударом. В этом цель эффекта мгновенной смерти. Хотя шанс срабатывания невелик, возможность мгновенного убийства остается силой, с которой следует считаться.

## **Расколдовывание**

Забудьте все заклинания снятия дополнительных статусов; почему бы не избавиться от всех болезней сразу! Эффект расколдовывания очищает все статусы целевого персонажа, независимо от того, хорошие они или плохие, и, хотя этот эффект может быть представлен как эффект снятия статуса, гораздо проще не пользоваться битовыми флагами.

## Телепортация

Хватит ходить, лучший способ путешествия — волшебная телепортация. Только РС могут использовать это заклинание. Телепортация может перенести РС в любое место на карте.

## Нацеливание заклинаний, стоимость и шансы

Эффект заклинания обычно нацелен на одного игрока, но не всегда. Иногда заклинание нацелено на того, кто его произносит, или на всех персонажей в заданной области. Кроме того, не все персонажи подвержены действию заклинания. Например, заклинание врага не должно поражать других врагов, а только игрока. Точно так же, заклинания, произнесенные игроком, должны быть нацелены только на врагов.

У каждого заклинания есть диапазон атаки; то есть заклинание может быть нацелено на любую цель, находящуюся внутри этого диапазона. Когда заклинание запущено и оказывает действие, у него есть определенная дистанция, на которую эффект заклинания распространяется наружу от точки удара. Заклинание, нацеленное на несколько персонажей, может повлиять на всех персонажей, находящихся в пределах действия эффекта заклинания.

Предположив, что персонаж знает заклинание (это диктуется отслеживанием битовой переменной для каждого персонажа), вы можете определить, сколько маны требуется для произнесения заклинания. Каждому заклинанию назначена определенная цена — персонаж должен иметь достаточно маны для произнесения заклинания. После сотворения заклинания его стоимость вычитается из очков маны произнесшего его персонажа.

Простое произнесение заклинания не означает, что оно сработает; есть шансы на ошибку. Шанс, что заклинание сработает или провалится называется шансом эффекта заклинания, и находится этот шанс в диапазоне от 0 процентов (никогда не работает) до 100 процентов (работает всегда).

## Главный список заклинаний

Каждый аспект заклинания, о котором вы прочитали, хранится в общей структуре, что делает работу с ними проще. Эта структура, **sSpell**, выглядит так:

```
typedef struct sSpell
{
    char Name[32];           // Название
    char Description[128];   // Описание

    long DmgClass;           // Класс, которому заклинание
                           // наносит двойной ущерб
    long CureClass;         // Класс, который заклинание лечит

    long Cost;              // Цена заклинания в МР
```

```
float Distance;           // Максимальное расстояние до цели

long Effect;              // Эффект заклинания
long Chance;              // % получения эффекта
float Value;              // Различные значения

long Target;              // Цель заклинания
float Range;              // Дистанция (в игровых единицах)

long MeshNum;             // Номер используемой сетки
long MeshPos;             // Позиционирование сетки
float MeshSpeed;          // Скорость перемещения сетки
long MeshSound;           // Звуковой эффект для воспроизведения
BOOL MeshLoop;            // Циклическая анимация
} sSpell;
```

---

**ПРИМЕЧАНИЕ** Структура **sSpell** определена во включаемом файле **msl.h**, расположенном в каталоге проекта **Char** (посмотрите в каталоге **\BookCode\Chap12\Chars** на прилагаемом к книге CD-ROM). Дополнительную информацию о проекте **Chars** вы найдете в конце главы.

---

Как видите, каждому заклинанию назначены название и описание, оба хранящиеся в небольших буферах. Ваш игровой движок будет отображать название каждого заклинания, предлагая игроку выбрать одно из заклинаний, когда придет время.

Ранее в этой главе я упоминал классы персонажей. Они оказывают эффект на заклинания. Отдельные заклинания могут наносить двойной ущерб слабо защищенным против них персонажам, и в этом причина появления переменной **sSpell::DmgClass**. Если класс персонажа и переменная **DmgClass** совпадают, заклинание наносит двойной ущерб.

С другой стороны, если класс персонажа совпадает с классом заклинания, заклинание может лечить персонаж. Представьте себе заклинание замораживания для ледяного дракона. Вместо того, чтобы повредить дракону, оно исцелит его на половину величины наносимого заклинанием ущерба. Таким образом, назначение **sSpell::CureClass** становится очевидным; если класс персонажа и **CureClass** совпадают, заклинание лечит, а не повреждает.

Перемещаясь дальше вы увидите стоимость произнесения заклинания (**sSpell::Cost**), измеряемую в очках маны. Чтобы сотворить заклинание, персонаж должен иметь в резерве как минимум указанное количество маны (**Cost**). После произнесения заклинания значение переменной **Cost** вычитается из маны персонажа.

Вспомните, что у заклинания есть назначенные диапазон и дистанция; диапазон (**sSpell::Range**) это расстояние от заклинателя, на котором заклинание может поразить цель, а дистанция (**sSpell::Distance**) — это размер области вокруг точки попадания, в которой имеет место эффект заклинания.

Когда заклинание находит свою цель, переменная **sSpell::Target** определяет на кого или на что будет оказан эффект — на самого заклинателя, на единственную цель, захваченную в параметре, или на всех персонажей, захваченных в параметре. Каждый тип цели определяется в движке так:

```
enum SpellTargets {
    TARGET_SINGLE = 0,
    TARGET_SELF,
    TARGET_AREA
};
```

Эффект заклинания (**sSpell::Effect**) имеет связанный с ним шанс на успех, хранящийся в **sSpell::Chance**. Каждое значение имеет в своем распоряжении трио переменных (**sSpell::Value**). Первое значение в массиве — это наносимое или излечиваемое количество повреждений, либо используемые битовые значения статусов.

Оставшиеся значения используются только для заклинаний с эффектом телепортации; для NPC и врагов первые три из этих значений являются координатами внутри текущего уровня, куда персонаж перемещается при произнесении заклинания телепортации. Что касается PC, четвертая переменная используется, чтобы указать на какую карту игрок должен переключиться когда заклинание произнесено. Из-за сложности телепортирования PC позвольте обрабатывать такие ситуации телепортирования скриптовому движку игры.

Завершающую группу переменных (**MeshNum**, **MeshPos**, **MeshSpeed**, **MeshSound** и **MeshLoop**) вы используете для графической составляющей заклинания. Вместо того, чтобы ссылаться на сетки заклинания по имени, более эффективно использовать номера. **MeshNum** хранит номер сетки, используемый движком управления заклинаниями для рисования графики заклинания.

**MeshPos** — это массив переменных, содержащих местоположение каждой сетки. Помните, что сетка может парить над заклинателем или целью, перемещаться от одного к другому, и даже растягиваться между двумя персонажами. Вы можете присвоить переменной **MeshPos** одно из следующих значений:

```
enum AnimPositions {
    POSITION_NONE = 0,
    POSITION_CASTER,
    POSITION_TOTARGET,
    POSITION_TOCASTER,
    POSITION_TARGET,
    POSITION_SCALE
};
```

И снова, с каждой сеткой связана скорость перемещения или время, в течение которого она отображается (когда она висит над персонажем или растягивается между двумя позициями). И скорость и время хранятся в переменной **MeshSpeed**, поскольку используется только одно из этих значений (в зависимости от перемещения сетки).



В вычислениях скорости **MeshSpeed** определяет расстояние в трехмерных единицах, которое сетка проходит за одну секунду. Для времени переменная **MeshSpeed** конвертируется в значение **long**, представляющее количество миллисекунд в течении которых сетка остается на месте.

Если сетка может завершить свой цикл анимации прежде, чем достигнет цели или прежде чем истечет время отображения, переменная **MeshLoop** говорит движку управления заклинаниями о необходимости повторять цикл анимации снова и снова, пока цикл сетки не завершится.

И, в качестве завершающего подарка, каждая из этих трех сеток имеет возможность воспроизводить звук при инициализации (позиционировании) сетки. Вообразите, что ваше заклинание шаровой молнии шипя летит к цели и из динамиков раскатывается звук взрыва! Вы ссылаетесь на каждый звук по номеру и позволяете вашему игровому движку воспроизводить эти звуки.

## Список заклинаний

Вы используете массив структур **sSpell** для хранения информации о каждом заклинании в игре. Этот массив структур называется главным списком заклинаний (далее мы будем ссылаться на него как на MSL) и хранится как последовательный файл данных. Структура данных заклинания относительно мала, и это значит, что список может быть полностью загружен в начале игры, для экономии времени при доступе к данным.

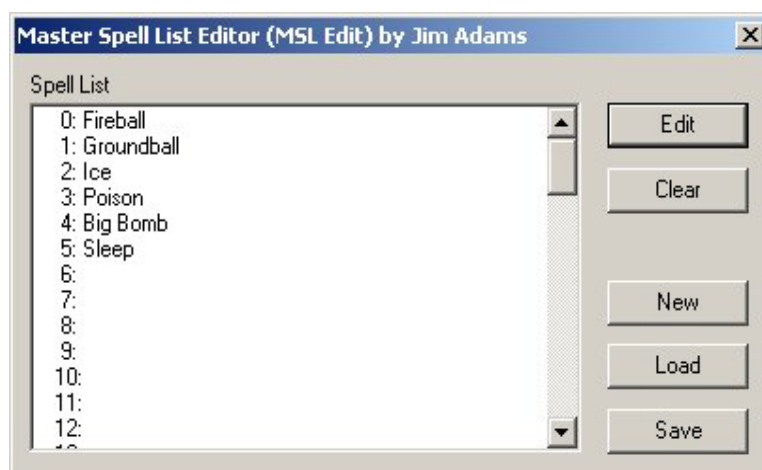
Оглянувшись назад, вы увидите, что я определил для каждого персонажа возможность использовать 64 заклинания, так что MSL должен хранить только 64 структуры данных **sSpell**, каждая из которых представляет отдельное заклинание, доступное для использования всеми персонажами.

Как я упоминал ранее, встает вопрос загрузки каждой структуры **sSpell** с соответствующими, необходимыми заклинанию данными. Даже когда в вашем распоряжении только 64 заклинания, попытка жестко прописать в коде столько структур данных потребует много работы.

### Определение заклинаний с MSL Editor

Определение заклинаний вашей игры путем ручного конструирования множества структур **sScript** быстро наскучит вам. вместо этого вам необходим редактор, который лучше подходит для быстрого изменения каждого аспекта заклинаний вашей игры. Встречайте MSL Editor!

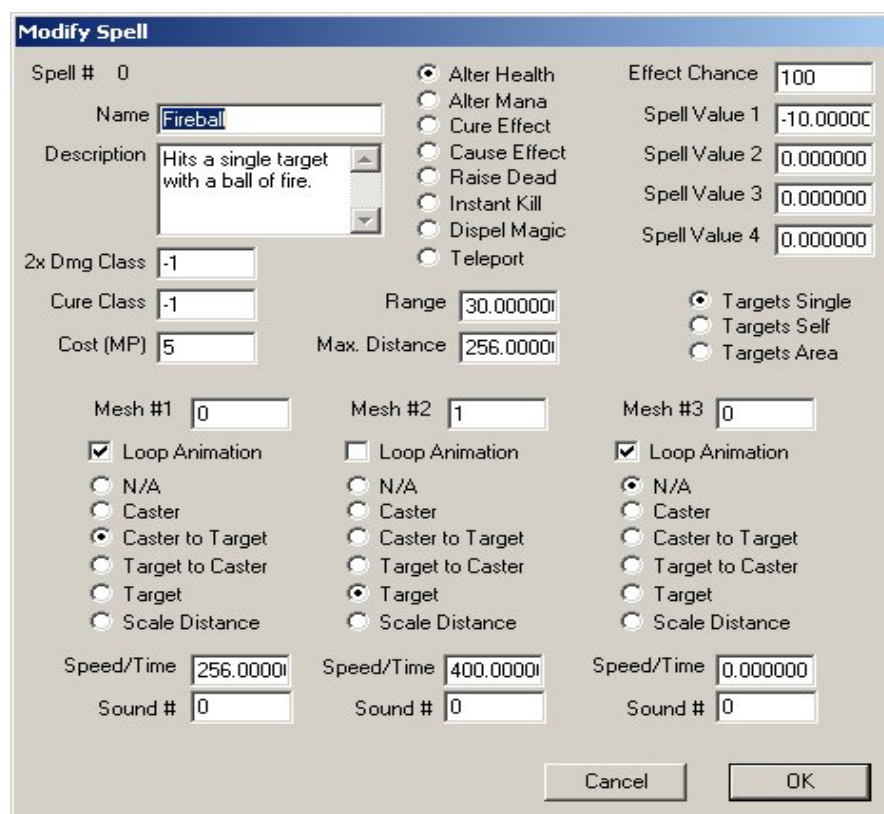
MSL Editor (находящийся на прилагаемом к книге CD-ROM в папке \BookCode\Chap12\MSLEdit), обладает прямолинейным интерфейсом, показанным на рис. 12.10.



*Рис. 12.10. Главный экран MSL Editor содержит список загруженных заклинаний, а также кнопки, управляющие добавлением, удалением, сохранением и загрузкой файлов MSL*

В MSL Editor предусмотрено место для 64 заклинаний (ограниченное только флагами, используемыми для хранения известных персонажу заклинаний). Запустив редактор вы можете выполнять следующие шаги для создания или редактирования ваших заклинаний:

1. Дважды щелкните по заклинанию в списке, чтобы открылось диалоговое окно **Modify Spell**.
2. В диалоговом окне **Modify Spell** (рис. 12.11) введите данные заклинания. Щелкните **OK**, чтобы закрыть диалоговое окно **Modify Spell** и вернуться к диалоговому окну **Master Spell List Editor**.



*Рис. 12.11. Диалоговое окно **Modify Spell** набито графикой и всей информацией, необходимой вам для описания эффектов заклинаний*

3. Чтобы сохранить ваш список заклинаний, щелкните по кнопке **Save**.
4. В диалоговом окне **Save MSL File** введите имя файла и щелкните **OK**. Чтобы загрузить файл заклинаний, щелкните по кнопке **Load** в диалоговом окне **Master Spell List Editor**, введите имя загружаемого файла заклинаний и щелкните **OK**.

Модификация заклинаний может сперва немного пугать, поскольку в нее вовлечены все данные, но ранее в этой главе вы изучили, что делает каждый фрагмент данных заклинания. Теперь, глядя на рис. 12.11, разберем небольшой пример определения показанного на рисунке заклинания шаровой молнии.

Заклинание шаровой молнии поражает единственную цель огненным шаром. Стоящее 5 MP, это заклинание не имеет назначенных классов, которые будет излечивать или наносить двойной ущерб (на это указывает значение -1). Заклинание изменяет здоровье на -10 (что указано в текстовом поле **Value1**), и имеет 100-процентную вероятность оказания на цель ожидаемого эффекта.

Целями заклинания могут быть персонажи, находящиеся в пределах 256 единиц от заклинателя, и заклинание поразит только одну цель (первого персонажа, найденного в области, размером 30 единиц).

Используются две сетки; первая сетка с номером 0 перемещается от заклинателя к цели со скоростью 256 единиц в секунду. Эта сетка циклически анимируется, пока не достигнет назначенной цели, и в этот момент ее место занимает вторая сетка. Вторая сетка имеет номер 1 и парит поверх цели 400 миллисекунд. Третья сетка не используется (для этого вы выбираете N/A в соответствующем поле). Сейчас заклинание завершено.

Выделите время и посмотрите примеры заклинаний, включенные в демонстрационную программу из этой главы (**Chars**). Попробуйте добавить некоторые собственные заклинания и проверьте их визуальные эффекты и функции на демонстрационных персонажах.

## Создание контроллера заклинаний

Управление заклинаниями — это вопрос отслеживания представляющих заклинание сеток и анимаций и последующая обработка эффектов заклинаний на назначенных целях. Поскольку эффекты заклинаний на самом деле связаны с персонажами, лучше позволить обрабатывать эффекты заклинаний движку, который управляет персонажами, и оставить объекту класса контроллера заклинаний анимацию заклинаний.

Вам надо создать класс контроллера заклинаний, поддерживающий список сотворяемых в данный момент заклинаний и отображающий их на экране. Когда заклинание завершено, класс контроллера вызывает внешнюю функцию для обработки эффектов заклинания.

Класс контроллера заклинаний **cSpellController** использует вспомогательные структуры, упрощающие отслеживание сеток и анимаций заклинаний. Это структуры **sSpellMeshList** и **sSpellTracker**.

---

**ПРИМЕЧАНИЕ** Класс **cSpellController**, структура **sSpellMeshList** и структура **sSpellTracker** находятся в файлах **spell.h** и **spell.cpp** на прилагаемом к книге CD-ROM (посмотрите в каталоге \BookCode\Chap12\Chars).

---

## Сетки и **sSpellMeshList**

Просмотрев структуру данных заклинания **sSpell**, вы увидите, что мы ссылаемся на сетки по номеру, а не по имени. Этот номер сетки является в действительности индексом в массиве сеток. Вы храните этот массив сеток в коллекции структур **sSpellMeshList**:

```
typedef struct sSpellMeshList {
    char        Filename[MAX_PATH]; // Имя файла сетки/анимации
    long        Count; // Кол-во использующих сетку заклинаний
    cMesh        Mesh; // Объект сетки
    cAnimation Animation; // Объект анимации

    // Конструктор и деструктор для подготовки и освобождения данных
    sSpellMeshList() { Count = 0; }
    ~sSpellMeshList() { Mesh.Free(); Animation.Free(); }
} sSpellMeshList;
```

Для каждой используемой в вашем движке сетки у вас есть соответствующая структура **sSpellMeshList**. Каждый экземпляр структуры хранит имя используемого файла сетки, объекты **cMesh** и **cAnimation** для сетки, и переменную (**Count**), которая отслеживает количество используемых в данный момент экземпляров сетки.

Для каждого заклинания, которому нужна сетка, соответствующий файл .X загружается в объекты сетки и анимации (оба используют одно и то же имя файла, а анимация использует единственный анимационный набор с именем **anim**).

Сетки загружаются с диска только когда они требуются контроллеру заклинаний, и поскольку структура поддерживает счетчик количества использований сетки, контроллер заклинаний может быстро определить, загружена ли сетка.

Когда заклинание завершает свой цикл анимации, счетчик использований соответствующей сетки уменьшается, и когда количество использующих сетку заклинаний сократится до нуля, объекты сетки и анимации освобождаются (для экономии памяти).

## Отслеживание заклинаний с использованием **sSpellTracker**

В то время, как структура **sSpellMeshList** поддерживает используемые заклинаниями сетки, сам список активных заклинаний поддерживается структурой **sSpellTracker**. Структура **sSpellTracker** выделяется и

вставляется в связанный список таких же структур каждый раз, когда произносится заклинание:

```
typedef struct sSpellTracker
{
    long SpellNum; // Номер заклинания

    sCharacter *Caster; // Персонаж, произнесший заклинание
    long      Type;     // Тип персонажа для воздействия

    long CurrentAnimation; // Анимация: 0-2
    float SourceX, SourceY, SourceZ; // Координаты источника
    float TargetX, TargetY, TargetZ; // Координаты цели

    float XPos, YPos, ZPos; // Текущие координаты
    float XAdd, YAdd, ZAdd; // Значения перемещения
    float Distance;        // Расстояние до цели

    union {
        float Speed; // Скорость перемещения
        long Time;   // Время показа
    };

    cObject Object; // Графический объект

    sSpellTracker *Prev, *Next; // Связанный список

    sSpellTracker() { Caster = NULL; Prev = Next = NULL; }
    ~sSpellTracker() { delete Next; }
} sSpellTracker;
```

Для каждого произнесенного заклинания структура **sSpellTracker** используется для хранения информации, отслеживающей сетку, анимацию, передвижение, время и какой персонаж произнес заклинание. Структура начинается с номера заклинания (**SpellNum**), который ссылается прямо на MSL.

Чтобы позднее помочь определить эффект заклинания, поддерживается также указатель на персонаж (**Caster**) и тип персонажей, на которые действует заклинание (PC, NPC или MC). Вы можете определить каждый тип персонажей следующим образом:

```
#define CHAR_PC      0
#define CHAR_NPC     1
#define CHAR_MONSTER 2
```

Заметьте, что в заклинании не определен целевой персонаж, но есть трио координат цели. Фактически, у заклинания есть трио координат источника. Помните, что сетка заклинания может неподвижно размещаться поверх заклинателя или жертвы, перемещаться между ними двумя или растягиваться между ними. Установка координат источника и цели гарантирует, что система отслеживания будет знать как позиционировать используемую сетку.

Говоря об используемых сетках, **CurrentAnimation** используется для того, чтобы отслеживать, какая из трех сеток используется. Как вы

помните, движение сетки может делиться на три этапа, и заклинение оказывает эффект только когда текущая анимация проходит третий этап.

Чтобы отслеживать перемещение сеток заклинаний (если они действительно двигаются), вы используете набор переменных (**XAdd**, **YAdd** и **ZAdd**), которые сообщают контроллеру заклинаний, в каком направлении перемещается сетка при каждом обновлении. Что касается используемого в данный момент местоположения сетки, переменные **XPos**, **YPos** и **ZPos** содержат текущие координаты, в которых визуализируется сетка.

Скорость, с которой сетка перемещается, хранится в **Speed**, а общее расстояние, которое должна пройти сетка, хранится в **Distance**. Если сетка стоит на месте, переменная **Time** служит обратным счетчиком, отсчитывающим время жизни сетки в миллисекундах.

Завершает **sSpellTracker Object** — графический объект, используемый для визуализации сеток, и **Prev** и **Next**, используемые для поддержки связанного списка структур.

## Класс *cSpellController*

Поскольку контроллер заклинаний необходим только для отслеживания сеток заклинаний и анимаций, определение класса относительно мало:

```
class cSpellController
{
private:
    cGraphics *m_Graphics; // Родительский графический объект
    cFrustum *m_Frustum; // Пирамида видимого пространства

    sSpell m_Spells[NUM_SPELL_DEFINITIONS]; // Данные заклинаний
    sSpellTracker *m_SpellParent; // Список активных заклинаний

    long m_NumMeshes; // Кол-во используемых сеток
    sSpellMeshList *m_Meshes; // Список сеток

    char m_TexturePath[MAX_PATH]; // Путь к текстурам сеток

    cCharacterController *m_Chars; // Контроллер персонажей

    // Установка перемещения сетки
    BOOL SetAnimData(sSpellTracker *SpellPtr, long Num);

    // Переопределяемая функция для воспроизведения звуков
    virtual BOOL SpellSound(long Num) { return TRUE; }
```

Прежде чем исследовать открытые функции, рассмотрим закрытые данные контроллера заклинаний. Контроллер заклинаний использует графический объект и объект пирамиды видимого пространства. Графический объект (**m\_Graphics**) должен быть заранее инициализирован для использования в классе, в то время как объект пирамиды видимого пространства (**m\_Frustum**) может предоставляться из внешнего кода или вычисляться внутри функции визуализации заклинаний.

Затем идет MSL, содержащий массив **m\_Spells**. Обратите внимание, что макроопределение **NUM\_SPELLDEFINITIONS** описывает размер массива MSL, а это значит, что вы легко можете подстроить размер для позднейших усовершенствований.

За MSL следует указатель связанного списка **m\_SpellParent**, отслеживающий заклинания, которые были произнесены и теперь отображаются. Далее идут **m\_NumMeshes** (где хранится количество используемых сеток) и **m\_Meshes** (список сеток).

Поскольку этот пример использует для представления заклинаний трехмерные сетки, вам необходимо загружать текстуры, и чтобы контроллер заклинаний мог найти эти текстуры, вы должны сохранить путь, указывающий местоположение растрового изображения, которое будет использовано как текстура.

Нечто, с чем вы сталкиваетесь впервые, это указатель **m\_Chars**, указывающий на используемый объект класса контроллера персонажей. Этот указатель на класс запускает эффекты заклинания (вы больше узнаете об этом вопросе в разделе «Создание класса контроллера персонажей» далее в этой главе).

Класс **cSpellController** содержит две закрытые функции: **SetAnimData** и **SpellSound**. Функция **SetAnimData** устанавливает используемую сетку, а также параметры перемещения сетки. **SpellSound** вызывается каждый раз, когда используется сетка заклинания; ваша задача переопределить эту функцию, чтобы воспроизводился соответствующий звук, как указано в списке аргументов функции.

Рассмотрев закрытые данные и функции, вы можете перейти к открытым функциям класса (конструктор, деструктор, **Init**, **Shutdown**, **Free**, **GetSpell**, **Add**, **Update** и **Render**):

```
public:
    cSpellController(); // Конструктор
    ~cSpellController(); // Деструктор

    // Функции для инициализации/выключения класса контроллера
    BOOL Init(cGraphics *Graphics, char *DefinitionFile,
             long NumSpellMeshes, char **MeshNames,
             char *TexturePath,
             cCharacterController *Controller);
    BOOL Shutdown();

    // Освобождение класса
    BOOL Free();

    sSpell *GetSpell(long SpellNum);

    // Добавление заклинания к списку
    BOOL Add(long SpellNum,
            sCharacter *Caster, long TargetType,
            float SourceX, float SourceY, float SourceZ,
            float TargetX, float TargetY, float TargetZ);
```

```
// Обновление всех заклинаний на основании
// прошедшего времени
BOOL Update(long Elapsed);

// Визуализация всех сеток заклинаний внутри
// пирамиды видимого пространства
BOOL Render(cFrustum *Frustum=NULL, float ZDistance=0.0f);
};
```

Поскольку глава становится очень длинной, я просто перечислю каждую функцию и опишу, что она делает. Чтобы следить за изложением вы можете загрузить код класса с прилагаемого к книге CD-ROM (загляните в \BookCode\Chap12\Chars\Spell.cpp).

### cSpellController::cSpellController и cSpellController::~sSpellController

Как обычно в классах C++, конструктор и деструктор очищают данные класса и освобождают все используемые ресурсы, соответственно. Деструктор для очистки данных полагается на отдельную функцию (функция **Shutdown**).

### cSpellController::Init и cSpellController::Shutdown

Перед использованием класса контроллера заклинаний вы должны инициализировать его. Завершив работу с классом вы вызываете **Shutdown** для освобождения ресурсов. Имея самый длинный список аргументов среди всех функций контроллера, **Init** получает следующие параметры:

- Указатель на ранее инициализированный графический объект (**cGraphics**).
- Имя файла MSL (**DefinitionFile**).
- Количество используемых сеток заклинаний (**NumSpellMeshes**).
- Массив с именами файлов .X, содержащих используемые для заклинаний сетки (**MeshNames**).
- Путь к каталогу текстур (**TexturePath**).
- Указатель на пока не рассматривавшийся класс контроллера персонажей (**Controller**).

### cSpellController::Free

Когда вы завершили работу с экземпляром класса контроллера заклинаний, но хотите повторно использовать его не отключая, вызовите **cSpellController::Free**. Функция **Free** освобождает список отслеживаемых заклинаний, а также список сеток.

### cSpellController::GetSpell

Внешнему коду может потребоваться доступ к MSL и **GetSpell** удовлетворяет эту потребность. По предоставленному номеру заклинания функция возвращает указатель на массив с загруженным MSL.



### **cSpellController::Add**

Теперь начинается настоящая забава! Функцию **Add** вы будете использовать больше всего, поскольку она инициализирует заклинание. Список аргументов включает номер заклинания в MSL (от 0 до 63), указатель на структуру сотворившего заклинание персонажа, тип целевого персонажа, координаты источника и цели.

### **cSpellController::SetAnimData**

Закрывающая функция **SetAnimData** инициализирует три используемые заклинанием сетки. Если одна из сеток не используется (на это указывает значение **POSITION\_NONE** в **MeshPos**), выполняется переход к следующей из трех сеток. После того, как использованы все три сетки, запускается эффект заклинания.

### **cSpellController::Update**

Заклинания должны перемещаться, обновлять свою синхронизацию и инициировать различные этапы отображения сеток. За все эти функции отвечает **Update**. Чтобы использовать **Update** просто передайте промежуток времени (в миллисекундах), прошедший с последнего вызова **Update** (или промежуток времени, на который вы хотите чтобы контроллер обновил заклинания).

### **cSpellController::Render**

Последняя из функций, **Render**, используется для визуализации сеток всех действующих заклинаний. Предоставление функции **Render** необязательных пирамиды видимого пространства и расстояния просмотра поможет изменить способ визуализации сеток.

## ***Определение жертвы и обработка эффекта заклинания***

Что происходит после того, как заклинание сработало и эффекты обработаны? Как я упоминал ранее, заклинания влияют только на персонажей, поэтому только движок управления персонажами должен менять данные персонажа. В разделе «Создание класса контроллера персонажей» далее в этой главе, вы увидите, как обрабатывать заклинания относительно персонажей.

## **Использование контроллера заклинаний**

К этому моменту контроллер заклинаний полнофункциональный, но без помощи контроллера персонажей контроллер заклинаний не работает. Однако, на минуту забудьте об этом и посмотрите следующий пример, показывающий как использовать контроллер заклинаний. Прежде чем создавать экземпляр контроллера заклинаний, объявим массив с именами файлов сеток:

```
// Graphics = ранее инициализированный объект cGraphics
// Используем две сетки
char *g_SpellMeshNames[] = {
    { "Fireball.x" },
    { "Explosion.x" }
};
```

Затем создадим экземпляр контроллера заклинаний и инициализируем его.

```
cSpellController Controller;

// Инициализация контроллера
Controller.Init(&Graphics, "default.msl",
               sizeof(g_SpellMeshNames)/sizeof(char*),
               g_SpellMeshNames, "..\\", NULL);
```

Теперь вы готовы к действиям. Предполагая, что у вас в MSL есть единственное заклинание (заклинание 0), вы можете активировать его с помощью следующего кода:

```
Controller.Add(0, NULL, CHAR_MONSTER,
               0.0f, 0.0f, 0.0f, 100.0f, 0.0f, 100.0f);
```

Заклинание теперь будет перемещаться от координат 0, 0, 0 к координатам 100, 0, 100, используя заданные в MSL Editor параметры. Когда закончите работу с контроллером заклинаний, убедитесь, что вызвали функцию контроллера **Shutdown**.

```
Controller.Shutdown();
```

## Сражения и персонажи

Есть моменты, когда персонажи просто не могут ужиться вместе. Тогда вы должны научить этих тварей, кто здесь босс. Для вашей игры необходима поддержка сражений, которую, к счастью, легко выполнить.

Хотя вы и ожидаете от боевых последовательностей роскошной графики и крутых эффектов, начинать следует с основ. В основе каждого сражения лежит набор правил (называемых *правилами сражения*, *combat rules*), определяющих последствия каждого взмаха оружием, каждого отклоненного удара, и результаты каждого магического заклинания.

Ранее в этой главе вы узнали о способностях персонажей — эти способности определяют силу персонажа, проворство и т.д. В данный момент вам наиболее интересны те способности, которые определяют, достигла ли атака намеченной цели и сколько ущерба она нанесла. Следуя краткому набору правил, вы можете использовать эти способности персонажей, чтобы определить исход сражения.

## Использование правил сражения для атаки

Ваша игра при обработке сражений сильно зависит от базового набора правил, точно так же, как и традиционные ролевые игры, разыгрываемые при помощи ручки и бумаги. Эти правила представляют собой набор математических формул, которые, будучи примененными с учетом случайности, определяют исход атаки, повреждения и защиту.

*Набор правил сражения (combat rule set, CRS)* вашей игры отталкивается от способностей, атрибутов и умений персонажа, которые вы уже видели в этой главе. Помните, как этим способностям, умениям и атрибутам назначались числовые значения? Догадываетесь зачем? Эти значения используются для генерации нескольких значений, определяющих исход боевых действий.

Например, атрибут меткости игрока используется в вычислении случайного числа, чтобы увидеть, куда попала атака. Затем сравнение с проворностью оппонента атакующего определяет, была ли атака отклонена. Если оппонент оказался не настолько удачлив, в игру вступает значение атаки атакующего, определяющее ущерб. Помните также, что у атакуемого персонажа также есть способность защиты, помогающая уменьшить количество повреждений.

После принятия решения об атаке, результат определяют несколько шагов.

### Нанесение удара

Когда персонаж наносит удар другому персонажу, это действие запускает процесс, определяющий, достиг ли удар цели. Успешность удара зависит от меткости атакующего персонажа и проворности атакуемого. Помните, что чем выше значение способности, тем больше шанс на поражение или отклонение атаки.

Значение меткости может находиться в диапазоне от 0 (всегда промахивается) до 999 (всегда попадает). Взяв случайное число и сравнив его со значением меткости вы можете быстро определить, произошло ли попадание. Если случайное число меньше или равно значению атрибута меткости, удар попал в цель. Следующий код показывает, как определить успешность удара:

```
// ToHit = значение атрибута меткости персонажа
long RandomValue = rand() % 1000;
BOOL HitFlag = (RandomValue <= ToHit) ? TRUE : FALSE;
```

В показанном выше коде **HitFlag** устанавливается в **TRUE**, если удар попал, или, скорее, если удар должен попасть. Для повышения шансов поражения цели у атакующего могут быть определенные дополнительные статусы, уменьшающие или увеличивающие значение меткости. Два используемых дополнительных статуса, которые влияют на меткость атакующего это слепота и орлиный глаз. Статус слепоты уменьшает

меткость на 25 процентов, а орлиный глаз повышает шансы попасть в цель на 50 процентов.

Чтобы применить любой модификатор дополнительного статуса, умножьте на него определенное значение меткости:

```
if(Ailments & AILMENT_BLIND)
    ToHit = (long)((float)ToHit * 0.75f);

if(Ailments & AILMENT_HAWKEYE)
    ToHit = (long)((float)ToHit * 1.5f);

long RandomValue = rand() % 999;
BOOL HitFlag = (RandomValue <= ToHit) ? TRUE : FALSE;
```

## Отклонение атаки

Помните, что с началом нападения в игру вступает проворность жертвы. Чем больше проворность защищающегося, тем больше у жертвы шансов отклонить атаку. Вы вычисляете отклонил ли защищающийся атаку точно также, как определяли, нанес ли атакующий удар:

```
// Agility = значение проворности персонажа
RandomValue = rand() % 999;
BOOL DodgeFlag = (RandomValue <= Agility) ? TRUE : FALSE;
```

### ВНИМАНИЕ!

Вы можете определить из вычислений способности уклоняться, что чем выше значение проворности, тем больше шанс уклониться от атаки. Поэтому в общем случае не надо устанавливать слишком большие значения для способности персонажа, поскольку он может стать неприкосновенным.

Для увеличения или уменьшения шансов отклонения атаки вы можете использовать дополнительные статусы неуклюжести и устойчивости. Неуклюжесть уменьшает шансы отклонить атаку на 25 процентов, а устойчивость повышает шансы на 50 процентов (это значит, что персонаж, у которого установлены оба эти дополнительные статусы, увеличивает свои шансы отклонить атаку на 25 процентов):

```
if(Ailments & AILMENT_CLUMSY)
    Agility = (long)((float)Agility * 0.75f);

if(Ailments & AILMENT_SUREFOOTED)
    Agility = (long)((float)Agility * 1.5f);

long RandomValue = rand() % 999;
BOOL DodgeFlag = (RandomValue <= Agility) ? TRUE : FALSE;
```

## Обработка ущерба

Когда определено, что удар поразил жертву, настает время вычислить, какой ущерб был нанесен, и здесь вступают в игру способности персонажей атаковать и защищаться. Ущерб обычно является переменным, и это значит,

что очень редко одна и та же атака наносит каждый раз одинаковое количество повреждений. И снова вы используете несколько случайных факторов.

Чтобы все осталось простым, вы можете взять значение способности атаковать атакующего (или, по крайней мере, от 90 до 110 процентов ее) и вычесть значение способности защищаться жертвы (от 80 до 100 процентов ее). Обратите внимание, что здесь также оказывают влияние дополнительные статусы и, наряду с ними, использование предметов увеличивающих способности атаковать и защищаться.

Верно. Предметы экипировки добавляют множители к способностям атаковать и защищаться. Ключом является значение модификатора предмета. Это значение представляет собой величину от 0 и больше, которая делится на 100 и увеличивается на единицу, что дает значение множителя, используемого совместно со значением способности. Например, оружие со значением модификатора 150 увеличивает способность атаковать на 50 процентов:

```
// Attack = значение способности атаковать персонажа
// Item[] = массив главного списка предметов
long Attack = (long)((float)Attack *
    (((float)Item[Weapon].Value / 100.0f) + 1.0f));
```

Возвращаясь к дополнительным статусам, на атаку и защиту влияют два из них — усиление и слабость. Слабость уменьшает атаку и защиту наполовину, в то время как усиление увеличивает значения на 50 процентов. Вот как работает все вместе для определения применяемого количества ущерба:

```
// Attack = значение способности атаковать атакующего
// Defense = значение способности защищаться обороняющегося
// Item[] = массив главного списка предметов
// Weapon = номер оружия в списке предметов (или -1, если его нет)
// Armor = номер доспехов в списке предметов (или -1, если их нет)
// Shield = номер щита в списке предметов (или -1, если его нет)

// Определяем количество атаки

// Начинаем с добавления модификатора имеющегося оружия
if(Weapon != -1)
    long Attack = (long)((float)Attack *
        (((float)Item[Weapon].Value / 100.0f) + 1.0f));

// Подстраиваем в соответствии с дополнительными статусами
if(Ailments & AILMENT_WEAK)
    Attack = (long)((float)Attack * 0.5f);
if(Ailments & AILMENT_STRONG)
    Attack = (long)((float)Attack * 1.5f);

// Определяем количество защиты

// Применяем модификаторы доспехов и щита
if(Armor != -1)
    Defense = (long)((float)Defense *
        (((float)Item[Armor].Value / 100.0f) + 1.0f));
```

```

if(Shield != -1)
    Defense = (long)((float)Defense *
        (((float)Item[Shield].Value / 100.0f) + 1.0f);

// Применяем дополнительные статусы
if(Ailments & AILMENT_WEAK)
    Defense = (long)((float)Defense * 0.5f);
if(Ailments & AILMENT_STRONG)
    Defense = (long)((float)Defense * 1.5f);

float DamagePercent = ((float)(rand() % 70) + 50.0f) / 100.0f;
long DamageAmount = (long)((float)Attack * DamagePercent);

// Определяем количество повреждений
// (используя здесь некоторые случайные коэффициенты)
float Range = (float)((rand() % 20) + 90) / 100.0f;
long DmgAmount = (long)((float)Attack * Range);
Range = (float)((rand() % 20) + 80) / 100.0f;
DmgAmount -= (long)((float)Defense * Range);

```

В конце концов переменная **DmgAmount** будет содержать количество повреждений, с которым и следует иметь дело. Однако, к этому моменту вы еще не закончили работу, поскольку теперь вступает в игру класс персонажа. Если атака сильно действует на персонажи данного класса, повреждения удваиваются. Если жертва имеет ту же природу, что и способ атаки, атака лечит жертву на половину вычисленного количества повреждений! Я позволяю вам самим поработать с этими вычислениями.

---

### ВНИМАНИЕ!

И снова, способность защищаться у персонажа не должна быть настолько высокой, что защищающийся персонаж будет очень редко получать какие-либо повреждения, когда атака достигает цели.

---

## Заклинания в битве

Теперь вы можете поместить все заклинания, которые ваша игра предлагает для использования. Вы знаете, как работают заклинания, но вам надо знать, как заклинания влияют на персонажи. Помните, что контроллер заклинаний отслеживает только сетки, создающие визуальную часть заклинания; эффект заклинания определяет контроллер персонажей.

Заклинания в битве используются в основном для нанесения повреждений врагам. Заклинания используют набор вычислений для определения итогового эффекта заклинания, также как это делается для физических атак. У заклинаний есть шанс неудачи, определяемый значением шанса заклинания в определении заклинания. Шанс срабатывания заклинания повышается способностью интеллекта персонажа, использующей следующие вычисления для определения множителя, применяемого к значению шанса:

```

// Chance = шанс срабатывания заклинания
// Mental = значение интеллекта заклинателя
Chance = (long)((float)Mental / 100.0f + 1.0f) * (float)Chance);

```

Последняя строка показывает, что значение интеллекта может находиться в диапазоне от 0 и выше. Значение 150 означает увеличение шансов на 50 процентов, а значение 200 означает удвоение шанса. Для помощи жертвам заклинаний у персонажей есть связанное с ними значение сопротивляемости, также выступающее как множитель:

```
// Resistance = сопротивляемость цели
Chance = (long)((1.0f - (float)GetResistance(Target) /
    100.0f) * (float)Chance);
```

Когда определено, что заклинание окажет эффект, предпринимаются соответствующие действия для обработки результата. Единственный эффект заклинания, с которым вы хотите бороться в это время, — повреждения. Всякий раз, когда жертве наносятся повреждения, способность сопротивляемости жертвы используется для сокращения наносимого ущерба. Сопротивляемость — это процентное значение, а это значит, что значение 0 не сокращает повреждения от заклинания, а значение 100 полностью отклоняет заклинание.

Дополнительные статусы также работают предусмотренным для них образом при произнесении заклинаний. Статус немоты означает, что игрок не может даже произнести магическое заклинание, а статус ошеломления наполовину сокращает интеллектуальные способности персонажа. И, наконец, состояния зачарованности и барьера наполовину сокращают сопротивляемость жертвы или увеличивают сопротивляемость на 50 процентов, соответственно.

Вы можете использовать следующий код, чтобы определить, оказало ли заклинание влияние на жертву, и как много ущерба было причинено:

```
// Chance = шанс срабатывания магического заклинания
// Mental = Интеллект сотворившего заклинание
// Resistance = Сопротивляемость жертвы
// Amount = Базовое количество наносимых заклинанием повреждений

// Применяем дополнительные статусы к интеллекту и сопротивляемости
if(Ailments & AILMENT_DUMBFOUNDED)
    Mental /= 2;
if(Ailments & AILMENT_ENCHANTED)
    Resistance = (long)((float)Resistance * 0.5f);
if(Ailments & AILMENT_BARRIER)
    Resistance = (long)((float)Resistance * 1.5f);

// Проверяем шанс срабатывания и вычисляем ущерб
Chance=(long)(((float)Mental / 100.0f + 1.0f) * (float)Chance);
if((rand() % 100) < Chance) {
    float Resist = 1.0f - ((float)Resistance / 100.0f);
    long DmgAmount = (long)((float)Amount * Resist);

    // Применяем дополнительные повреждения или исцеления,
    // зависящие от класса
}
```

После того, как заклинание поразит свою цель, вычисляется соответствующее количество повреждений для применения. Помните, что

для отдельных классов заклинания могут вызывать двойной ущерб, по сравнению с заклинанием обычного персонажа, в то время как другие заклинания могут излечивать на половину наносимого ущерба.

Поскольку вы создадите реальный контроллер персонажей в разделе «Создание класса контроллера персонажей» позже в этой главе, отложим до него завершение работы с заклинаниями.

## Интеллект в битвах

Хотя игроки способны полностью управлять их персонажами в игре, на вашу долю остается управление NPC. Чтобы игра была достойной, искусственный интеллект ваших NPC должен быть равным противником для сражений. Их действия должны подражать вашим, выбирают ли они атаку, исцеляют себя или произносят заклинание.

Персонажам дают рудиментарный интеллект, когда они вступают в битву. Если персонаж потерял более половины своего здоровья, или находится под действием дополнительного статуса, он может попытаться исцелить себя или снять статус. Это означает, что он ищет в своем списке известных заклинаний (если он есть) и произносит подходящее заклинание для лечения.

Если, с другой стороны, РС попадает в область видимости другого персонажа, враждебный персонаж выбирает, напасть ли физически или провести магическую атаку (если известны какие-нибудь заклинания). Вам необходимо назначить шансы, что персонаж проведет нападение, для каждого типа атаки. Обратите внимание, что атаки базируются на встроенном заряде атакующего существа — чтобы создание атакowało, его заряд должен быть полным.

Когда принято решение атаковать ближайший персонаж, выбирается либо физическая атака, либо магическое заклинание. Произносятся только заклинания, которые вредят другим персонажам. Если в радиусе доступности нет живого целевого персонажа, игра случайным образом решает, что персонаж пытается зачаровать себя, используя заклинания установки дополнительных статусов, таким образом увеличивая свою силу, проворство или другие полезные качества.

Специфика выполнения предшествующих действий вступает в игру, когда вы создаете контроллер персонажей, который будет принимать такие решения за ваших персонажей. Мы обсудим это позже, в разделе «Создание класса контроллера персонажей».

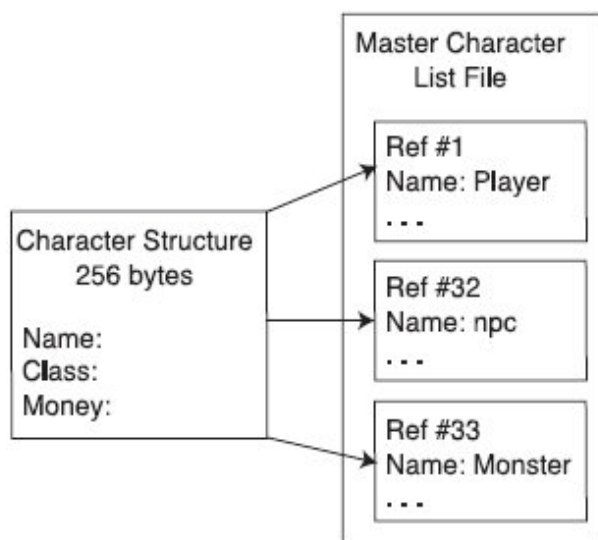
## Построение главного списка персонажей

Вы создаете и используете главный список персонажей (MCL) во многом также, как вы создаете и используете главный список предметов (MIL) для определения объектов в вашей игре. Перед тем, как использовать их в вашей игре, вам надо спроектировать каждый персонаж, вместе с его



представлением (трехмерной сеткой) и функциональностью (способностями и атрибутами). Информация о персонаже хранится в структуре **sCharacterDefinition**.

MCL хранится таким же образом, как и MIL, в последовательном файле данных (рис. 12.12). Всякий раз, когда в игре нужен персонаж, ссылаются на MCL; каждому персонажу назначен уникальный номер для ссылки. Когда персонаж необходим, вы загружаете соответствующую структуру данных.



*Рис. 12.12. Файл MCL разделен на секции (структуры), все одинакового размера, что делает проще позиционирование и загрузку конкретной структуры по ее ссылочному номеру*

Теперь взглянем на структуру **sCharacterDefinition**:

```
typedef struct sCharacterDefinition
{
    // Различные данные
    char   Name[32];      // Название персонажа
    long   Class;         // Номер класса персонажа
    long   Money;         // Количество денег
    float  Speed;         // Скорость перемещения
    long   MagicSpells;   // Битовые флаги для отметки
                          // известных заклинаний
    long   MeshNum;       // Номер сетки для загрузки

    // Способности
    long   Agility;       // Проворность
    long   Attack;        // Атака
    long   Defense;       // Защита
    long   Resistance;    // Сопротивляемость магии
    long   Mental;        // Интеллект

    // Атрибуты
    long   HealthPoints;  // Кол-во очков здоровья (максимум)
    long   ManaPoints;    // Кол-во очков маны (максимум)
    long   ToHit;         // Шансы попадания
    long   Level;         // Уровень опыта
    long   Experience;     // Очки опыта

    // Имущество
    char   ItemFilename[MAX_PATH]; // Имя файла CharICS
    long   Weapon;         // Оружие из экипировки
    long   Armor;          // Доспех из экипировки
}
```

```

long Shield;                // Щит из экипировки
long Accessory;             // Аксессуар из экипировки

// Данные бросаемого предмета
long DropChance; // Вероятность выпадения предмета при гибели
long DropItem;   // Номер выпадающего при гибели предмета

// Шансы и эффекты атаки/магии
float Range;      // Дистанция атаки
float ChargeRate; // Обратный счетчик частоты атак
long ToAttack;    // Вероятность атаки
long ToMagic;     // Вероятность использования магии
long EffectChance; // Вероятность осуществления эффекта атаки
long Effects;     // Битовые флаги эффекта атаки
} sCharacterDefinition;

```

---

**ПРИМЕЧАНИЕ** Вы найдете структуру **sCharacterDefinition** во включаемом файле **mcl.h**, находящемся в исходных кодах программы Chars на прилагаемом к книге CD-ROM (загляните в \BookCode\Chap12\Chars).

---

Точно также, как главный список предметов, MCL хранит только минимум информации о персонаже. Поскольку в игровом мире одновременно могут существовать несколько персонажей одного типа (например, 10 гоблинов), данные конкретного экземпляра хранятся отдельно. Эти данные экземпляра включают координаты персонажа, текущие значения очков здоровья и очков манны и т.д.

Структура **sCharacterDefinition** хранит шаблон, используемый, когда создается экземпляр персонажа. Этот шаблон охватывает все персонажи, включая PC.

Хотя структура хорошо прокомментирована, смысл некоторых вещей может быть сразу не понятен. В дополнение к способностям и атрибутам, о которых вы уже читали, у вас есть различные данные, имущество, бросаемый предмет, шансы и эффекты атаки/магии. Таблица 12.6 описывает, что эти переменные делают в определении персонажа.

**Таблица 12.6.** Различные переменные структуры sCharacterDefinition

Переменная	Описание
<b>Name</b>	Название персонажа (ограничено 32 байтами, включая символ завершения строки).
<b>Class</b>	Номер класса персонажа. Персонажам назначены классы и персонаж может использовать только предметы, помеченные как относящиеся к этому же классу. Также отдельные атаки и заклинания по-разному действуют на разные классы.
<b>Money</b>	Количество денег, которое несет персонаж.

**Таблица 12.6.** Различные переменные структуры `sCharacterDefinition` (продолжение)

<i>Переменная</i>	<i>Описание</i>
<b>MagicSpells</b>	Массив из двух переменных <b>long</b> , содержащих битовые флаги, определяющие заклинания, известные персонажу. Начиная с первого байта в массиве, младший бит представляет заклинание 0, второй бит представляет заклинание 1, и так, пока не будет достигнут шестьдесят четвертый бит, представляющий заклинание 63.
<b>MeshNum</b>	Поддерживается внешний массив сеток, который определяет сетки и анимации, используемые для персонажей. Значение, хранимое в <b>MeshNum</b> , это индекс используемой сетки в этом массиве.
<b>ItemFilename</b>	Это имя файла системы управления имуществом персонажа, используемого для PC. NPC используют ICS только если они собираются торговать с игроком.
<b>Weapon</b>	Индекс в MIL оружия, которым в данный момент экипирован персонаж. Если значение <b>-1</b> , предмет экипировки отсутствует (то же самое применимо к последующим предметам экипировки).
<b>Armor</b>	Индекс в MIL доспехов, которыми в данный момент экипирован персонаж.
<b>Shield</b>	То же, что <b>Weapon</b> и <b>Armor</b> . Индекс в списке предметов щита из текущей экипировки.
<b>Accessory</b>	Завершает набор индекс в списке предметов аксессуара из текущей экипировки персонажа
<b>DropChance</b>	Процент вероятности того, что убитый персонаж выронит предмет.
<b>DropItem</b>	Номер бросаемого предмета, используемый, когда персонаж убит и определено (через <b>DropChance</b> ), что он выронил предмет.
<b>Range</b>	У персонажей есть обычный диапазон атаки, распространяющийся до самого удаленного края ограничивающего объема. Значение диапазона атаки, чтобы быть эффективным, должно быть больше нуля.
<b>ChargeRate</b>	После того, как персонаж выполнил действие, такое как атака, использование предмета, экипирование предметом или произнесение заклинания, он некоторое время не может выполнять другие действия. <b>ChargeRate</b> задает скорость обратного отсчета времени, по истечении которого персонаж сможет выполнять другие действия

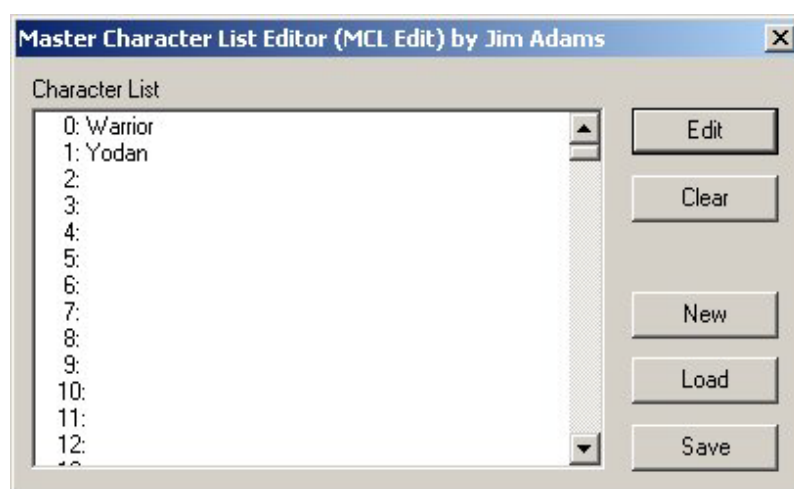
Таблица 12.6. Различные переменные структуры sCharacterDefinition (окончание)

Переменная	Описание
<b>ToAttack</b>	У персонажей есть два варианта атаки — физическая и магическая. <b>ToAttack</b> задает процент вероятности того, что персонаж, получив шанс, использует физическую атаку.
<b>ToMagic</b>	<b>ToMagic</b> задает процент вероятности того, что, получив шанс, персонаж произнесет заклинание.
<b>EffectChance</b>	Если персонаж атакует, это процент вероятности того, что будет произведен ожидаемый магический эффект.
<b>Effects</b>	Эффекты атаки персонажа. Это набор битовых флагов, и вы решаете, что означает каждый разряд.

Сконфигурировать единственный персонаж так же просто, как заполнить анкету, но когда надо описать 100 персонажей, все быстро усложняется. Вам нужен MCL Editor.

## MCL Editor

К этому моменту вы, вероятно, уже пользовались редакторами, и этот столь же удобен, что и остальные. Если вы еще не сделали этого, запустите приложение MCLEdit (вы найдете его на прилагаемом к книге CD-ROM в каталоге \BookCode\Chap12\MCLEdit). На рис. 12.13 показано диалоговое окно MCL Editor.



*Рис. 12.13. Диалоговое окно **Master Character List Editor** содержит список персонажей и несколько кнопок, которые позволяют вам добавлять новый персонаж к списку, очищать определение персонажа, редактировать, сохранять и загружать список персонажей*

Работа с MCL Editor подобна работе с MSL Editor, обсуждавшимся ранее в этой главе, и с MIL Editor, обсуждавшимся в главе 11. MCL Editor может поддерживать до 256 персонажей, нумеруемых от 0 до 255. Каждый персонаж отображается в списке. Работая с MCL Editor, придерживайтесь следующих шагов:

1. Дважды щелкните по персонажу в списке или щелкните по кнопке **New**, чтобы было выведено диалоговое окно **Modify Character**, показанное на рис. 12.14.

*Рис. 12.14. Диалоговое окно **Modify Character** заполнено различной информацией о персонаже, хранящейся в структуре **sCharacterDefinition***

2. В диалоговом окне **Modify Character** введите в каждое из полей соответствующую информацию о персонаже. Вы можете менять название персонажа, класс, очки здоровья, очки манны, уровень, опыт, деньги, значения способностей, известные заклинания и сетку.
3. Закончив заполнять информацию о персонаже в диалоговом окне **Modify Character**, щелкните **OK**. Вы вернетесь к диалоговому окну **Master Character List Editor**.
4. Щелкните по кнопке **Save**, чтобы перейти к диалоговому окну **Save MCL File**.
5. Введите имя файла и щелкните по кнопке **Save**, чтобы записать файл MCL на диск.
6. Чтобы загрузить файл, щелкните по кнопке **Load** (в диалоговом окне **Master Character List Editor**), введите имя файла и щелкните **OK**.

Вы уже читали о каждом фрагменте описания персонажа. Теперь пришло время ввести эту информацию в соответствующие места. Когда дело касается заклинаний, подсвеченные номера отмечают те заклинания, которые персонаж знает автоматически. Эти номера заклинаний напрямую относятся к вашему MSL, так что вы можете одновременно запустить MSL Editor и MCL Editor и расположить их окна рядом для сравнения информации.

## Использование определений персонажей

Определение персонажа является по своей природе шаблоном, так что вам в действительности нужно загрузить определения и затем работать с ними на поэкземплярной основе. Это значит, что вы должны придумать контроллер, который загружает определения и отслеживает каждый экземпляр персонажа в вашей игре. Вам необходим класс контроллера персонажей.

## Создание класса контроллера персонажей

Теперь, увидев что вовлечено в управление и описание ваших игровых персонажей, можно сосредоточиться на конструировании класса контроллера, который будет заботиться обо всем за вас, включая добавление, удаление, обновление и визуализацию персонажей, а также обрабатывать эффекты заклинаний из ранее разработанного контроллера заклинаний.

Поскольку к отслеживанию персонажей относится так много всего, наша задача разделить все на несколько структур и один класс. Подобно заклинаниям, требуется список сеток для хранения списка используемых сеток. Однако на этот раз информация о заикливании анимации не хранится в определении персонажа; нужна другая структура для хранения анимаций персонажей, которые надо заиклить.

Для работы искусственного интеллекта вы создаете отдельную структуру для хранения координат маршрутных точек. И, наконец, еще одна структура поддерживает связанный список используемых персонажей. Теперь исследуем каждую из упомянутых структур и содержащуюся в них информацию.

## Сетки и sCharacterMeshList

Ранее, в разделе «Сетки и sSpellMeshList» вы прочитали о контроллере заклинаний и том, как контроллер поддерживает список сеток. Для контроллера персонажей у вас также есть список сеток, используемых для визуализации персонажей. Структура **sCharacterMeshList** содержит объекты анимации и сетки и имя файла.

```
typedef struct sCharacterMeshList {
    char Filename[MAX_PATH]; // Имя файла сетки/анимации
    long Count;               // Кол-во использующих сетку персонажей
    cMesh Mesh;               // Объект сетки
    cAnimation Animation;     // Объект анимации

    sCharacterMeshList() { Count = 0; }
    ~sCharacterMeshList() { Mesh.Free(); Animation.Free(); }
} sCharacterMeshList;
```

## Заикливание анимации и sCharAnimationInfo

Анимации, используемые персонажами, либо заиклены, либо нет. Некоторые действия, такие как ожидание, требуют, чтобы анимация сетки постоянно

повторялась, создавая впечатление постоянного движения, в то время как другие анимации, такие как взмах мечом, надо исполнять только один раз.

Храня список анимаций, которые необходимо зациклить, контроллер персонажа может передать эту информацию графическому ядру, чтобы оно выполнило за вас всю тяжелую работу. Вы храните эту информацию о зацикливании анимации в структуре **sCharAnimationInfo**, выглядящей так:

```
typedef struct {
    char Name[32]; // Имя анимации
    BOOL Loop;     // Флаг зацикливания
} sCharAnimationInfo;
```

Чтобы использовать структуру вы должны сохранить имя анимации (соответствующее имени анимационного набора в файле .X) и флаг, говорящий, надо ли зацикливать соответствующую анимацию. Вы узнаете об этом больше в разделе «Использование sCharacterController» далее в этой главе.

## Перемещение и sRoutePoint

Как обсуждалось ранее, вы используете структуру **sRoutePoint** для хранения координат маршрутных точек, через которые должны проследовать персонажи в их нескончаемом перемещении по уровню.

```
typedef struct {
    float XPos, YPos, ZPos; // Позиция цели
} sRoutePoint;
```

## Отслеживание персонажей с sCharacter

Теперь все станет сложнее, поскольку в отслеживание каждого персонажа вовлечено гораздо больше данных. Фактически в отслеживание каждого персонажа вовлечено так много данных (в структуре **sCharacter**), что нам придется рассматривать их по частям:

```
typedef struct sCharacter
{
    long Definition; // Номер определения персонажа
    long ID;         // Идентификатор персонажа

    long Type;       // PC, NPC или MONSTER
    long AI;         // STAND, WANDER и т.д.

    BOOL Enabled;    // Флаг активности (для обновлений)
```

Для начала каждому персонажу необходимо определение, которое берется из главного списка персонажей. Вы храните номер этого определения в переменной **Definition**. Чтобы различать похожих персонажей вы назначаете каждому персонажу уникальный идентификационный номер (**ID**). Думайте об использовании идентификаторов так же, как об использовании имен. Вместо того, чтобы в

ходе игры добавить персонаж с именем Джордж, вы ссылаетесь на тот же персонаж, как на персонаж 5.

Каждый отслеживаемый персонаж относится к определенному типу — PC, NPC или враг. Для определения значения переменной **Type** используются следующие три макроса:

```
#define CHAR_PC      0
#define CHAR_NPC     1
#define CHAR_MONSTER 2
```

Далее следуют параметры искусственного интеллекта персонажа. Вспомните, что персонаж может стоять на месте, бродить по уровню, ходить по заданному маршруту, следовать за другим персонажем или избегать другого персонажа. Тип искусственного интеллекта каждого персонажа хранится в переменной **AI** и для него может быть выбрано одно из следующих, заданных макросами, значений:

```
#define CHAR_STAND  0
#define CHAR_WANDER 1
#define CHAR_ROUTE  2
#define CHAR_FOLLOW 3
#define CHAR_EVADE  4
```

И, наконец, каждому персонажу нужно разрешение обновления. Это определяет флаг **Enabled** и, если он равен **TRUE**, контроллеру разрешено обновлять персонаж в каждом кадре, в то время как значение флага **FALSE** означает, что персонаж никогда не обновляется (пока вы не разрешите).

Вам необходимо хранить определение персонажа из MCL для ссылок, а для персонажей с имуществом еще и ICS. Следующие переменные структуры хранят эту информацию вместе с именем файла скрипта, который вызывается, когда игрок активирует персонаж:

```
sCharacterDefinition Def;      // Загруженное определение
cCharICS *CharICS;            // ICS персонажа
char ScriptFilename[MAX_PATH]; // Назначенный скрипт
```

Поскольку определение персонажа хранит только максимальные значения способностей и атрибутов, структуре **sCharacter** необходим способ отслеживать текущие значения, меняющиеся по ходу игры. В это включаются очки здоровья, очки маны, флаги дополнительных статусов и текущий заряд персонажа.

```
long  HealthPoints; // Текущие очки здоровья
long  ManaPoints;   // Текущие очки маны
long  Ailments;     // Дополнительные статусы персонажа
float Charge;       // Заряд атаки
```

Поскольку персонажи перемещаются вокруг, выполняя различные действия (перемещение, ожидание, атака и т.д.), вам необходимо обеспечить способ отслеживания их действий и местоположения. Кроме того, необходимо сохранять последнюю известную анимацию (для обновления анимации персонажа), а также время последнего обновления анимации.



**ПРИМЕЧАНИЕ** Помните, что отдельные действия (например, атаку или произнесение заклинания) персонажи могут выполнять только когда их заряд достигнет максимального значения. Этот заряд увеличивается со скоростью, заданной в MCL.

---

Вы используете следующие переменные структуры, чтобы отслеживать действия персонажа, координаты, направление движения и данные анимации:

```
long Action;           // Текущее действие
float XPos, YPos, ZPos; // Текущие координаты
float Direction;       // Направление персонажа
long LastAnim;         // Последняя анимация
long LastAnimTime;     // Последнее время анимации
```

Вы также должны предоставить способ, предотвращающий обновление персонажей, выполняющих определенные действия, пока это действие не будет завершено. Например, когда персонаж атакует, нет никакой необходимости продолжать обновление персонажа, пока он не закончит размахивать оружием. Необходим обратный счетчик времени для блокировки действий персонажа; этот обратный счетчик — **ActionTimer**.

Для постоянного прекращения обновления персонажа вы используете вторую переменную с именем **Locked**. Если вы присвоите **Locked** значение **TRUE**, контроллер персонажа не будет обновлять персонаж, пока вы не установите **Locked** в **FALSE**.

Вы определяете **ActionTimer** и **Locked** в структуре **sCharacter** так:

```
BOOL Locked;           // Блокировка определенных действий
long ActionTimer;      // Таймер обратного отсчета времени блокировки
```

Следующий набор переменных заботится о боевой стороне персонажей:

```
sCharacter *Attacker;   // Атакующий персонаж (если есть)
sCharacter *Victim;     // Атакуемый персонаж

long SpellNum;          // Заклинание для произнесения
                        // при готовности
long SpellTarget;       // Тип цели заклинания
float TargetX, TargetY, TargetZ; // Координаты цели заклинания
```

Когда один персонаж атакует другого, указатели на атакующий персонаж и на жертву сохраняются в их соответствующих структурах **sCharacter**. Атакующий запоминает жертву, а жертва запоминает атакующего. Также, когда персонаж использует заклинание, сохраняется номер заклинания в MSL вместе с координатами цели заклинания и типом персонажа-цели (**CHAR\_PC**, **CHAR\_NPC** или **CHAR\_MONSTER**).

Вспомните, что у персонажей есть действия, и у этих действий есть набор связанных с ними анимаций. Причина сохранения атакующего, жертвы и информации о заклинании (так же как и последующей информации о предмете), в том что действие персонажа и анимация должны быть

завершены прежде чем результат действия будет иметь место. Как только атакующий персонаж взмахнет оружием, будут вычислены результаты атаки.

То же самое относится к заклинаниям; как только персонаж произносит заклинание, информация о заклинании из структуры **sCharacter** используется для определения на кого или что было оказано воздействие. То же самое касается использования предметов; указатель на используемый предмет сохраняется в течение действия использования предмета вместе с указателем на структуру ICS персонажа **cCharItem** (для удаления предмета, если он помечен как **USEONCE**).

```
long      ItemNum;    // Предмет, используемый при готовности
sCharItem *CharItem; // Предмет для удаления из имущества
```

Мы на полпути через структуру. Теперь вы сохраняете информацию об искусственном интеллекте персонажа. Вы уже читали почти обо всех последующих данных. У вас есть расстояние до преследуемого или избегаемого персонажа вместе с указателем на преследуемый или избегаемый персонаж.

Для персонажей, которые используют ограниченную область, вы сохраняете минимальные и максимальные координаты, за которыми следует информация о маршруте:

```
float Distance;           // Расстояние преследования/избегания
sCharacter *TargetChar;   // Преследуемый персонаж
float MinX, MinY, MinZ;   // Минимальные ограничивающие координаты
float MaxX, MaxY, MaxZ;   // Максимальные ограничивающие координаты

long      NumPoints;      // Кол-во точек в маршруте
long      CurrentPoint;   // Текущая точка маршрута
sRoutePoint *Route;       // Маршрутные точки
```

Далее располагается трио переменных, используемых для хранения короткого сообщения, которое накладывается поверх персонажа в процессе игры (как показано на рис. 12.15):

```
char      Message[128]; // Текстовое сообщение
long      MessageTimer; // Таймер текстового сообщения
D3DCOLOR MessageColor; // Цвет текстового сообщения
```



*Рис. 12.15. Механизм сообщений, как показано здесь, отображает результат определенных действий. Персонаж, атаковавший другого, определяет, что намеченная жертва уклонилась от атаки*

Сообщения персонажа помогают передать немного информации, как показано на рис. 12.15. Чтобы установить сообщение, скопируйте строку сообщения (до 128 символов) в буфер **Message**, установите период времени (в миллисекундах), в течение которого будет отображаться сообщение, и назначьте цвет отображаемого текста.

Завершает класс **sCharacter** переменная объекта графического ядра **cObject**, которая поддерживает сетку и анимацию персонажа. Для улучшения визуального представления персонажа отдельный объект и сетка используются для представления оружия персонажа. Эта сетка оружия и объект конфигурируются каждый раз, когда персонаж экипируется новым оружием. Последними идут указатели связанного списка **Prev** и **Next**:

```
cObject Object;           // Объект класса персонажа
cMesh WeaponMesh;         // Сетка оружия
cObject WeaponObject;      // Объект оружия

sCharacter *Prev, *Next;  // Связанный список персонажей
```

Для каждого персонажа хранится много информации, и чтобы помочь контроллеру в подготовке структуры каждый раз, когда новый персонаж вступает в драку, структура **sCharacter** комплектуется конструктором и деструктором для подготовки данных и помощи в освобождении ресурсов:

```
sCharacter()
{
    Definition = 0; // Устанавливаем определение #0
    ID = -1;       // Нет ID
    Type = CHAR_NPC; // Персонаж NPC
    Enabled = FALSE; // Не активен

    Ailments = 0; // Нет доп. статусов
    Charge = 0.0f; // Нет заряда

    // Очищаем определение
    ZeroMemory(&Def, sizeof(sCharacterDefinition));
    CharICS = NULL; // Нет ICS

    ScriptFilename[0] = 0; // Нет скрипта

    Action = CHAR_IDLE; // Устанавливаем анимацию по умолчанию
    LastAnim = -1;      // Сбрасываем анимацию

    Locked = FALSE; // Не заблокирован
    ActionTimer = 0; // Нет таймера действия

    Attacker = NULL; // Нет атакующего
    Victim = NULL;   // Нет жертвы

    ItemNum = 0; // Нет предмета для использования
    CharItem = NULL; // Нет имущества

    Distance = 0.0f; // Задаем расстояние
    TargetChar = NULL; // Нет целевого персонажа

    // Очищаем ограничивающий прямоугольник
    // (для ограничения перемещений)
```

```

    MinX = MinY = MinZ = MaxX = MaxY = MaxZ = 0.0f;

    NumPoints = 0;          // Нет маршрутных точек
    Route = NULL;           // Нет маршрута

    Message[0] = 0;         // Очищаем сообщение
    MessageTimer = 0;       // Сбрасывает таймер сообщения

    Prev = Next = NULL;    // Очищаем указатели связанного списка
}

~sCharacter()
{
    if(CharICS != NULL) { // Освобождаем ICS персонажа
        CharICS->Free();
        delete CharICS;
    }
    delete [] Route;      // Освобождаем маршрут

    WeaponObject.Free();  // Освобождаем объект оружия
    WeaponMesh.Free();    // Освобождаем сетку оружия
    Object.Free();        // Освобождаем объект персонажа

    delete Next;         // Удаляем следующий персонаж в списке
}
} sCharacter;

```

Вот и все! Я говорил вам, что структура **sCharacter** большая, но она ничто в сравнении с использующим эту структуру классом контроллера персонажей.

## Класс sCharacterController

Мозг, оперирующий с персонажами, — это класс **cCharacterController** (содержащийся в файлах Chars.h и Chars.cpp), возможно, самый большой класс неигрового ядра, с которым вы будете работать. Из-за ограниченного пространства, вместо того чтобы показать определение класса здесь, я поместил код класса на прилагаемый к книге CD-ROM (смотрите в \BookCode\Chap12\Chars).

Класс **cCharacterController** поддерживает список активных персонажей, каждый из которых хранится в структуре **sCharacter**. Для каждого типа персонажей есть соответствующий элемент в массиве структур **sCharacterMeshList** (и соответствующая структура **sCharAnimationInfo**).

В начале файла Char.h расположено макроопределение:

```

// Количество персонажей в файле
#define NUM_CHARACTER_DEFINITIONS 256

```

За этим определением следуют макросы, которые вы уже видели, — типы персонажей, типы искусственного интеллекта и дополнительные статусы. Макроопределения после этого трио вы еще не видели, но к данному моменту должны понимать их назначение; это действия, которые

может выполнять персонаж (и соответствующие анимации). Взгляните на макроопределения:

```
// Типы персонажей
#define CHAR_PC          0
#define CHAR_NPC        1
#define CHAR_MONSTER    2

// Типы AI
#define CHAR_STAND       0
#define CHAR_WANDER     1
#define CHAR_ROUTE      2
#define CHAR_FOLLOW     3
#define CHAR_EVADE      4

// Дополнительные статусы
#define AILMENT_POISON      1
#define AILMENT_SLEEP      2
#define AILMENT_PARALYZE   4
#define AILMENT_WEAK       8
#define AILMENT_STRONG     16
#define AILMENT_ENCHANTED  32
#define AILMENT_BARRIER   64
#define AILMENT_DUMBFOUNDED 128
#define AILMENT_CLUMSY     256
#define AILMENT_SUREFOOTED 512
#define AILMENT_SLOW       1024
#define AILMENT_FAST       2048
#define AILMENT_BLIND      4096
#define AILMENT_HAWKEYE    8192
#define AILMENT_SILENCED   16384

// Типы действий/анимаций
#define CHAR_IDLE         0
#define CHAR_MOVE         1
#define CHAR_ATTACK       2
#define CHAR_SPELL        3
#define CHAR_ITEM         4
#define CHAR_HURT         5
#define CHAR_DIE          6
#define CHAR_TALK         7
```

Все остальное составляет класс контроллера.

```
class cCharacterController
{
private:
    cGraphics *m_Graphics; // Родительский графический объект
    cFont      *m_Font;    // Используемый объект шрифта
    cFrustum   *m_Frustum; // Пирамида видимого пространства
```

Начинают класс его закрытые данные с указателем на родительский объект **cGraphics** и указателем на объект **cFont** (используемый для рисования текста на экране). Вы должны заранее инициализировать оба объекта, прежде чем контроллер персонажей будет использовать их. Пирамиду видимого пространства вы используете так же, как делали это в контроллере заклинаний.

Далее идут имя файла MCL, указатели на MIL и MSL, и, наконец, указатель на контроллер заклинаний:

```
char m_DefinitionFile[MAX_PATH]; // Имя файла определений

sItem *m_MIL; // Главный список предметов
sSpell *m_MSL; // Главный список заклинаний

cSpellController *m_SpellController; // Контроллер заклинаний
```

По мере того, как персонажи добавляются к игре, счетчик (**m\_NumCharacters**) отслеживает их общее количество. За счетчиком следует указатель на родительскую (корневую) структуру **sCharacter** в связанном списке структур:

```
long m_NumCharacters; // Кол-во персонажей в списке
sCharacter *m_CharacterParent; // Список персонажей
```

Вы используете список структур сеток и анимаций так же, как применяли их в контроллере заклинаний. На этот раз, в дополнение к сохранению пути к текстурам, вы также создаете путь к каталогу где расположены сетки. Почему используется каталог сеток? В случае присоединения оружия к персонажу структура **sItem** хранит только имя файла, но не путь. Это означает, что сетки оружия должны находиться в том же каталоге, что и сетки персонажей.

```
long m_NumMeshes; // Кол-во используемых сеток
sCharacterMeshList *m_Meshes; // Список сеток
char m_MeshPath[MAX_PATH]; // Путь к сеткам оружия
char m_TexturePath[MAX_PATH]; // Путь к текстурам сеток

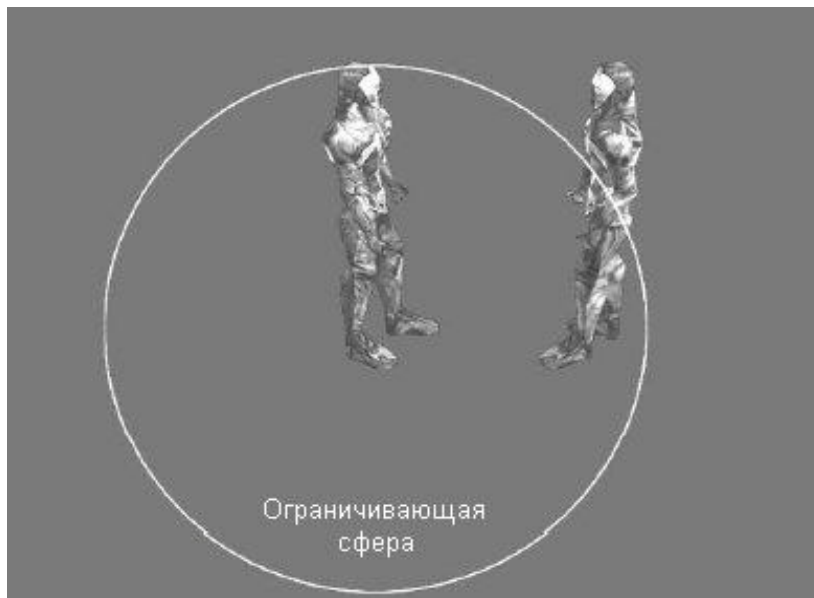
long m_NumAnimations; // Кол-во анимаций
sCharAnimationInfo *m_Animations; // Данные анимаций
```

На этом внутренние данные класса **sCharacterController** завершаются. Теперь вы можете переключить внимание на закрытые функции. Вы используете первую функцию, **GetXZRadius**, для вычисления максимального ограничивающего радиуса вдоль осей X и Z.

```
// Возвращает радиус X/Z персонажа
float GetXZRadius(sCharacter *Character);
```

Вы используете радиус X/Z, чтобы увеличить надежность обнаружения столкновения ограничивающих сфер. Чтобы увидеть, что я имею в виду, взгляните на рис. 12.16.

Более высокие персонажи в игре имеют нежелательный побочный эффект наличия больших ограничивающих сфер. Чтобы исправить это, при вычислении размера ограничивающей сферы используются только самые далекие точки персонажа по осям X и Z, поскольку эти оси представляют ширину и глубину персонажа, а не высоту.



*Рис. 12.16. Персонаж слева — высокий парень. Из-за его высоты ограничивающая сфера объекта персонажа распространяется далеко в каждом направлении, вызывая ложные результаты проверки столкновений, когда два персонажа встречаются*

Продолжая разбор функций вы встретите виртуальную функцию, используемую для воспроизведения звуковых эффектов, каждый раз при начале каких-либо действий. Ваша задача — выполнить наследование от класса **cCharacterController** и переопределить функцию, чтобы она делала что-нибудь полезное:

```
// Переопределяемая функция для воспроизведения звуков
virtual BOOL ActionSound(sCharacter *Character)
{ return TRUE; }
```

Следующие две функции вступают в работу совместно с функциональностью обновления класса. В каждом кадре, где необходимо обновление персонажа, вызывается заданная функция для обновления действий персонажа. Заданная функция зависит от типа обновляемого персонажа; для PC заданная функция это **PCUpdate** (которая переопределяется вами для управления вашим игроком).

Для NPC и врагов функция обновления называется **CharUpdate**. **PCUpdate** в данный момент ничего не делает, поскольку вы должны написать код для управления игроком в главном приложении. Для врагов и NPC уже заданы параметры их AI, так что контроллер знает, как обрабатывать их в **CharUpdate**.

```
// Функция перемещения для персонажа игрока
// (надо переопределить)
virtual BOOL PCUpdate(
    sCharacter *Character, long Elapsed,
    float *XMove, float *YMove, float *ZMove)
{ return TRUE; }

// Функция обновления персонажа
// для всех независимых персонажей
BOOL CharUpdate(sCharacter *Character, long Elapsed,
    float *XMove, float *YMove, float *ZMove);
```

Шаги, предпринимаемые чтобы инициировать обновление и обработать все действия персонажей, достаточно длительны. Во-первых, вы перебираете всех персонажей из списка активных персонажей. Вызываются соответствующие функции обновления персонажей (**PCUpdate** или **CharUpdate**).

После того, как определено, какое действие каждый персонаж собирается выполнять, должна быть проверена допустимость данного действия. Перемещающиеся вокруг персонажи не могут проходить сквозь других игроков (за исключением РС, которые могут проходить через других РС). Также, в зависимости от вашей высоты, вы используете функцию определения пересечения персонажа с картой. Эти две проверки выполняют следующие функции:

```
// Проверка допустимости перемещений.
// Контроль границ для других персонажей и вызов
// ValidateMove (переопределяемой).
BOOL CheckMove(sCharacter *Character,
               float *XMove, float *YMove, float *ZMove);

// Виртуальная ValidateMove для сверки внешних границ
// с перемещениями персонажа
virtual BOOL ValidateMove(sCharacter *Character,
                        float *XMove, float *YMove, float *ZMove)
{ return TRUE; }
```

Обе из показанных функций (**CheckMove** и **ValidateMove**) берут указатель на обновляемый персонаж, а также дистанцию перемещения персонажа по каждой из осей. Каждая функция соответствующим образом модифицирует эти значения. Когда персонаж перемещается и действие признано допустимым, другая функция является оберткой для действия и действительного обновления местоположения персонажа и действия.

```
// Завершение перемещения путем установки направления,
// анимации и т.д.
BOOL ProcessUpdate(sCharacter *Character,
                  float XMove, float YMove, float ZMove);
```

Всякий раз, когда персонажи сражаются друг с другом, некоторые из них умирают. Контроллер может быстро обработать смерть NPC и врагов, исключив соответствующие структуры из списка. Но, если дело касается РС, смерть может означать завершение игры, так что ее обработкой должно заниматься главное приложение. В этом причина появления **PCDeath**, которая получает единственный аргумент — указатель на гибнущий РС.

```
// Обработка смерти персонажа игрока
virtual BOOL PCDeath(sCharacter *Character)
{ return TRUE; }
```

Если говорить о смерти персонажей, каждый раз, когда умирает враг, есть шанс, что он выронит предмет или имеющиеся у него деньги. Поскольку все предметы на карте обрабатывает ваше главное приложение, ваша задача определить, выронил ли враг предмет или золото и добавить



соответствующий предмет к списку предметов карты. Переопределение следующих двух функций поможет вам каждый раз, когда враг роняет что-нибудь, получить точные координаты места его гибели и количество выроненных денег.

```
// Функции для роняния денег и предметов при гибели персонажа
virtual BOOL DropMoney(float XPos, float YPos, float ZPos,
                      long Quantity)
{ return TRUE; }

virtual BOOL DropItem(float XPos, float YPos, float ZPos,
                     long ItemNum)
{ return TRUE; }
```

Вы почти достигли конца долгого пути. Мы закончили с закрытыми данными и функциями, и нам остались открытые функции:

```
public:
    cCharacterController(); // Конструктор
    ~cCharacterController(); // Деструктор

    // Функции для инициализации/отключения класса контроллера
    BOOL Init(cGraphics *Graphics, cFont *Font,
             char *DefinitionFile, sItem *MIL, sSpell *MSL,
             long NumCharacterMeshes, char **MeshNames,
             char *MeshPath, char *TexturePath,
             long NumAnimations, sCharAnimationInfo *Anims,
             cSpellController *SpellController);
    BOOL Shutdown();
```

За типовыми конструктором и деструктором класса идет пара функций **Init** и **Shutdown**. Чтобы контроллер заработал вы должны сперва инициализировать его, вызвав **Init**. Когда вы завершаете работу с классом контроллера персонажей, вызовите в свою очередь **Shutdown**.

Здесь много аргументов, но каждый из них понятен. У вас есть родительский графический объект и объект шрифта, за которыми следует имя файла с определениями MCL. Далее идут указатели на главный список предметов и MSL. Помните, что MSL поддерживается контроллером заклинаний, так что для получения указателя на список необходимо вызвать **sSpellController::GetSpell**.

Затем идет количество используемых сеток персонажей вместе со списком сеток, путь к каталогу сеток и путь к каталогу текстур. Завершает список аргументов функции **Init** количество устанавливаемых структур циклов анимации, соответствующий указатель на массив структур анимации и указатель на используемый объект класса контроллера заклинаний.

Несколько похожая по природе на выключатель, следующая функция **Free** полностью удаляет всех персонажей из списка активных персонажей. Эта функция полезна для очистки списка, когда персонаж покидает уровень и в список необходимо поместить полностью новый набор персонажей:

```
// Освобождение класса
BOOL Free();
```

Если говорить о добавлении персонажей к списку, вот функция, которая делает все необходимое для этого:

```
// Добавление персонажа к списку
BOOL Add(long IDNum, long Definition, long Type, long AI,
         float XPos, float YPos, float ZPos,
         float Direction = 0.0f);

// Удаление персонажа из списка
BOOL Remove(long IDNum);
BOOL Remove(sCharacter *Character);
```

Для функции **Add** вам необходимо предоставить уникальный идентификационный номер, номер используемого определения персонажа в MCL, назначаемый персонажу тип (**CHAR\_PC**, **CHAR\_NPC** или **CHAR\_MONSTER**), используемый искусственный интеллект, координаты персонажа и угол относительно оси Y, ориентирующий персонаж в заданном направлении.

За **Add** идут две функции, удаляющие персонаж из списка. Первая версия функции **Remove** получает в качестве аргумента уникальный идентификационный номер персонажа, а вторая версия функции **Remove** получает указатель на структуру персонажа.

Обратите внимание, что я продолжаю говорить об удалении персонажей из списка. Что насчет всех усилий, которые вы вложили в вашего PC — как сохранить их достижения для последующей загрузки? Со следующим набором функций сохранения и загрузки, конечно!

```
// Сохранение и загрузка отдельного персонажа
BOOL Save(long IDNum, char *Filename);
BOOL Load(long IDNum, char *Filename);
```

Обе показанные выше функции получают идентификационный номер сохраняемого или загружаемого персонажа, а также имя используемого файла.

На этом заканчиваются функции, предназначенные чтобы подготавливать, добавлять и удалять персонажи игры. Настало время позволить им перемещаться вокруг, выполняя разные действия. Раньше вы видели функции, используемые для обновления отдельных типов персонажей; теперь появляется единая функция, которую вы вызываете для одновременного обновления всех персонажей:

```
// Обновление всех персонажей
// на основе прошедшего времени
BOOL Update(long Elapsed);
```

Функция **Update** вызывается в каждом кадре один раз. Получая единственный аргумент (время, прошедшее с последнего обновления), функция **Update** вызывает соответствующую функцию обновления каждого персонажа, проверяет допустимость перемещений и действий каждого персонажа, и завершается обработкой действий. Затем вызывается

**Render**, отображающая все видимые персонажи, находящиеся в заданной пирамиде видимого пространства.

```
// Визуализируем все объекты внутри
// пирамиды видимого пространства
BOOL Render(long Elapsed = -1,
            cFrustum *Frustum = NULL,
            float ZDistance = 0.0f);
```

У **Render** есть несколько необязательных аргументов. Первый из них вы используете чтобы управлять синхронизацией анимации персонажей. В средах с переключением задач, таких как Windows, простое использование времени, прошедшего с обработки последнего кадра недопустимо; вместо этого вы должны определить фиксированный период прошедшего времени и гарантировать, что игровой движок будет придерживаться обновлений с заданной частотой. В этой книге я обычно использую частоту обновлений в 30 кадров в секунду (задержка в 33 миллисекунды между кадрами).

Что касается указателя на пирамиду видимого пространства, приложение может предоставить собственный предварительно созданный объект, либо передать **NULL** (и необязательное расстояние по оси Z), чтобы класс создал свою собственную пирамиду видимого пространства.

Всякий раз, когда персонаж нуждается в обновлении, визуализации или еще чем-то, необходим указатель на связанный список персонажей для перебора элементов списка. Или, возможно, ваше приложение нуждается в доступе к данным персонажа. В любом случае вам помогут следующие функции:

```
// Получение структуры sCharacter
sCharacter *GetParentCharacter();
sCharacter *GetCharacter(long IDNum);
```

В определенных функциях перед тем как атаковать или произносить заклинание необходимо убедиться, что персонажи (такие, как враги) могут видеть других персонажей. В вашем проекте необходима функция, проверяющая, что линия взгляда чиста. Возвращаемое значение **TRUE** означает, что один персонаж находится в поле видимости другого.

```
// Проверяет линию взгляда для атаки/заклинания
virtual BOOL LineOfSight(
    sCharacter *Source, sCharacter *Target,
    float SourceX, float SourceY, float SourceZ,
    float TargetX, float TargetY, float TargetZ)
{ return TRUE; }
```

Когда контроллер персонажей (или внешний код) нуждается в одной из способностей персонажа, их необходимо получать с помощью следующей группы функций. Эти функции учитывают дополнительные модификаторы, такие как дополнительные статусы и предметы экипировки:

```
// Функции для получения настроенных способностей
float GetSpeed(sCharacter *Character);
long GetAttack(sCharacter *Character);
```

```

long  GetDefense(sCharacter *Character);
long  GetAgility(sCharacter *Character);
long  GetResistance(sCharacter *Character);
long  GetMental(sCharacter *Character);
long  GetToHit(sCharacter *Character);
float GetCharge(sCharacter *Character);

```

Следом идет огромный набор функций, которые вы используете для получения и установки отдельных сведений о персонаже (относящихся к функционированию искусственного интеллекта или действиям):

```

// Установка указателя на ICS персонажа
сCharICS *GetICS(long IDNum);

// Установка блокировки и таймера действия
BOOL SetLock(long IDNum, BOOL State);
BOOL SetActionTimer(long IDNum, long Timer);

// Установка расстояния избегания/следования
BOOL SetDistance(long IDNum, float Distance);
float GetDistance(long IDNum);

// Установка маршрутных точек
BOOL SetRoute(long IDNum,
              long NumPoints, sRoutePoint *Route);

// Установка скрипта
BOOL SetScript(long IDNum, char *ScriptFilename);
char *GetScript(long IDNum);

// Установка флага разрешения
BOOL SetEnable(long IDNum, BOOL Enable);
BOOL GetEnable(long IDNum);

// Функции для перемещения и получения координат персонажа
BOOL Move(long IDNum, float XPos, float YPos, float ZPos);
BOOL GetPosition(long IDNum,
                 float *XPos, float *YPos, float *ZPos);

// Функции для установки/получения
// границ перемещения персонажа
BOOL SetBounds(long IDNum,
               float MinX, float MinY, float MinZ,
               float MaxX, float MaxY, float MaxZ);
BOOL GetBounds(long IDNum,
               float *MinX, float *MinY, float *MinZ,
               float *MaxX, float *MaxY, float *MaxZ);

// Функции для установки/получения типа персонажа
BOOL SetType(long IDNum, long Type);
long GetType(long IDNum);

// Функции для установки/получения AI персонажа
BOOL SetAI(long IDNum, long Type);
long GetAI(long IDNum);

// Установка целевого персонажа
BOOL SetTargetCharacter(long IDNum, long TargetNum);

```

Опуская детали предыдущих функций (обратитесь к разделу «Перемещение персонажей» за информацией об их функциональности), вы

сталкиваетесь с функцией, используемой для установки данных, которые будут отображаться в сообщении поверх персонажа:

```
// Установка текста сообщения, плавающего над персонажем
BOOL SetMessage(sCharacter *Character, char *Text,
               long Timer, D3DCOLOR Color=0xFFFFFFFF);
```

**SetMessage** позволяет вам временно наложить строку текста на **Timer** миллисекунд, рисуя текст заданным цветом. Вы устанавливаете сообщения персонажа для информирования игрока о событиях, таких как количество потерянных из-за атаки очков здоровья.

Далее следует функция, обрабатывающая ущерб, полученный от физической или магической атаки (это определяется флагом **PhysicalAttack**; значение **TRUE** соответствует физической атаке, а **FALSE** — магической):

```
// Обработка ущерба, нанесенного заклинанием
// или физической атакой
BOOL Damage(sCharacter *Victim,
            BOOL PhysicalAttack, long Amount,
            long DmgClass, long CureClass);
```

**Damage** получает указатель на персонаж, которому причинен ущерб, тип ущерба (физический или магический), количество наносимого ущерба и классы, которым эта атака наносит двойной ущерб или исцеляет. Вы подстраиваете размер ущерба на основании защиты и сопротивляемости жертвы.

Получив достаточно повреждений, персонажи умирают, и, когда это происходит, вызывается следующая функция:

```
// Обработка смерти NPC/врага
BOOL Death(sCharacter *Attacker, sCharacter *Victim);
```

Получая указатель на жертву, контроллер может соответствующим образом обработать ее смерть. Если жертва является врагом, вы используете указатель на атакующего персонажа, чтобы добавить ему очки опыта. Также, в случае смерти врага, функция **Death** определяет сколько золота выронил враг и какой предмет (если он есть) он уронил и вызывает соответствующие функции контроллера для обработки потери этих предметов.

Предваряя следующую функцию напомним, что всякий раз когда РС убивает врага, этот РС получает опыт, хранящийся в описании врага в MCL. Чтобы применить этот опыт используйте следующую функцию:

```
// Обработка увеличения опыта
virtual BOOL Experience(sCharacter *Character,
                      long Amount)
{ Character->Def.Experience += Amount; return TRUE; }
```

Обратите внимание, что функция **Experience** может быть переопределена. Это может происходить, когда вы используете отдельный

движок боевых последовательностей; вы не хотите добавлять опыт РС, пока битва не будет закончена. Соответственно вы используете свою собственную функцию чтобы отслеживать, сколько опыта добавить, когда сражение окончено.

Переопределение функции также может иметь место, когда персонажу нужно повышать уровень опыта, как только он наберет заданное количество очков опыта. Функция **Experience** — это место, где определяется, когда персонажу надо повысить уровень опыта, и предпринимаются соответствующие действия по увеличению его способностей.

Еще одно замечание о функции **Experience**: контроллер персонажей обычно отображает количество очков опыта, получаемых игроком при убийстве врага. Чтобы прекратить отображение контроллером этой информации (как в случае отдельных боевых последовательностей), возвращайте из функции **Experience** значение **FALSE**.

Следующая пара функций отвечает за обработку атак и заклинаний. Обе функции получают указатель на атакующий персонаж (если он есть), а также его намеренную жертву. Для заклинаний требуется структура **sSpellTracker**, чтобы сообщить контроллеру какое заклинание обрабатывать, а также структура **sSpell**, содержащая информацию об эффектах используемого заклинания:

```
// Обработка физической атаки от атакующего к жертве
BOOL Attack(sCharacter *Attacker, sCharacter *Victim);

// Обработка последствий заклинания, когда оно завершено
BOOL Spell(sCharacter *Caster,
           sSpellTracker *SpellTracker, sSpell *Spells);
```

Каждая из предыдущих функций учитывает способность персонажа к атаке и защите и соответственно подстраиает значения. Когда сражающиеся соединяются, обрабатываются повреждения. Когда обнаруживается, что заклинание повлияло на цель (помните, что есть вероятность неудачи), вызывается следующая функция для обработки эффектов:

```
// Применение эффектов заклинания
BOOL SpellEffect(sCharacter *Caster, sCharacter *Target,
                sSpell *Spell);
```

В этой точке контроллера постепенно исчезают вещи. Вы используете следующие функции для экипировки, использования и бросания предметов:

```
// Обработка экипировки/разэкипировки предмета
BOOL Equip(sCharacter *Character, long ItemNum,
           long Type, BOOL Equip);

// Обработка использования предмета персонажем
BOOL Item(sCharacter *Owner, sCharacter *Target,
          long ItemNum, sCharItem *CharItem = NULL);

// Обработка выбрасывания предмета
BOOL Drop(sCharacter *Character,
          sCharItem *Item, long Quantity);
```

Для **Equip** вы должны указать модифицируемый персонаж и номер предмета (из **MIL**), который включается в экипировку. Вы используете аргумент **Type**, чтобы задать тип предмета экипировки (**WEAPON**, **ARMOR**, **SHIELD** или **ACCESSORY**) и флаг **Equip**, чтобы сообщить контроллеру, надо ли экипировать персонаж указанным предметом (**Equip** равно **TRUE**) или удалить из экипировки используемый в данный момент предмет (установив **Equip** в **FALSE**).

Для функции использования предмета (**Item**) требуются два персонажа: владелец предмета и персонаж, для которого этот предмет используется. Благодаря этому один персонаж может использовать лечебный эликсир для другого персонажа. Укажите номер используемого предмета в **MIL**, а также указатель на структуру **CharItem** ICS владельца, чтобы могло быть уменьшено количество предметов.

Следующая функция необходима для обработки эффекта заклинания телепортации РС. Когда для РС используется заклинание телепортации, контроллер персонажей вызывает следующую функцию для обработки эффекта. Ей передаются указатели на целевой персонаж и структуру заклинания:

```
// Обработка заклинания телепортации РС
virtual BOOL PCTeleport(sCharacter *Character,
                       sSpell *Spell)
{ return TRUE; }
```

Завершает класс контроллера персонажей функция, отвечающая за подготовку персонажа к выполнению действия. Вы используете эту функцию в основном когда управляете вашим РС через функцию **PCUpdate**:

```
// Установка действия (с таймером)
BOOL SetAction(sCharacter *Character,
               long Action, long AddTime = 0);
};
```

Когда РС (как, впрочем, и любой персонаж) делает что-нибудь, выполняется соответствующее действие. Ходьба — это действие, атака — это действие и т.д. Ранее эти действия были определены как **CHAR\_IDLE**, **CHAR\_MOVE**, **CHAR\_ATTACK** и т.д. Вам необходимо задать в аргументе **Action** одно из этих значений для инициации действия персонажа.

Для каждого действия, которое может выполнять персонаж, есть соответствующая анимация в массиве структур **sCharAnimationInfo**, используемом для инициализации контроллера. Когда персонаж выполняет действие, устанавливается соответствующая анимация, а также таймер действия, используемый для обратного отсчета времени, необходимого для завершения анимации. Помните, что никакие дальнейшие действия не могут быть выполнены, пока не завершено текущее действие.

Последний аргумент в списке, **AddTime**, применяется для добавления дополнительных миллисекунд к таймеру действия. Задание для **AddTime** значения **-1** заставляет **SetAction** не использовать таймер действия, а это значит, что действие будет очищено при следующем обновлении.

## Использование sCharacterController

Вы получили множество функций в классе **cCharacterController**, и хотя вы уже прочитали об их функциональности, трудно представить, что же каждая из них делает. Возможно, поможет пример.

Начнем с установки сеток и информации об анимации для сетки каждого персонажа. Данный пример использует две сетки:

```
char *g_CharMeshNames[] = {
    { "..\\Data\\Warrior.x" }, // Сетка # 0
    { "..\\Data\\Yodan.x" }   // Сетка # 1
};
```

Каждая сетка содержит список анимаций, представляющих действия, которые может выполнять каждый персонаж. Каждая анимация действия в двух сетках совместно использует одни и те же имена анимационных наборов. Вы отображаете эти имена, используя структуру **sCharAnimationInfo** (которая хранит имя анимации в файле .X, а также флаг, определяющий, зациклена ли анимация) следующим образом:

### ВНИМАНИЕ!

Заметьте, что в структурах данных анимации я повторно использовал некоторые анимации, что исключительно здорово. Просто убедитесь, что вы не устанавливаете флаг зацикливания анимации в **TRUE**, если позже устанавливаете флаг зацикливания в **FALSE**, иначе все не будет работать. То же самое применимо, когда вы сперва устанавливаете для другого действия флаг зацикливания в **FALSE**, а позже устанавливаете его в **TRUE**.

Например, следующий фрагмент кода правильный:

```
sCharAnimationInfo Anims[] = {
    { "Idle", TRUE }, { "Walk", FALSE }
};
```

В то время как следующий фрагмент не работает (поскольку вы меняете флаг зацикливания у одной и той же используемой анимации):

```
sCharAnimationInfo Anims[] = {
    { "Idle", TRUE }, { "Idle", FALSE }
};
```

```
sCharAnimationInfo g_CharAnimations[] = {
    { "Idle", TRUE }, // Действие CHAR_IDLE
    { "Walk", TRUE }, // Действие CHAR_MOVE
    { "Swing", FALSE }, // Действие CHAR_ATTACK
    { "Spell", FALSE }, // Действие CHAR_SPELL
    { "Swing", FALSE }, // Действие CHAR_ITEM
    { "Hurt", FALSE }, // Действие CHAR_HURT
};
```



```
{ "Die", FALSE }, // Действие CHAR_DIE
{ "Idle", TRUE } // Действие CHAR_TALK
};
```

Теперь, когда вы определили, какие сетки используются и как их анимировать, вы можете инициализировать контроллер персонажей и начать добавлять персонажи:

```
// Graphics = ранее инициализированный объект cGraphics
// Font = ранее инициализированный объект шрифта
// MIL = массив главного списка предметов (sItem MIL[1024])
// SpellController = ранее созданный контроллер заклинаний

cCharacterController Controller;

// Инициализация контроллера
Controller.Init(&Graphics, &Font, "..\\Data\\Default.mcl",
               (sItem*)&MIL, SpellController.GetSpell(0),
               sizeof(g_CharMeshNames)/sizeof(char*), g_CharMeshNames,
               "..\\Data\\", "..\\Data\\",
               sizeof(g_CharAnimations) / sizeof(sCharAnimationInfo),
               (sCharAnimationInfo*)&g_CharAnimations,
               &SpellController);

// Добавляем NPC (определение в MCL #0), который бродит
// внутри области от -256,0,-256 до 256,0,256
Controller.Add(0, 0, CHAR_NPC, CHAR_WANDER,
              0.0f, 0.0f, 0.0f, 0.0f);
Controller.SetBounds(0, -256.0f, 0.0f, -256.0f,
                    256.0f, 0.0f, 256.0f);
```

Теперь, когда вы добавили к списку NPC, вы можете непрерывно обновлять и визуализировать его в каждом кадре:

```
long UpdateCounter = timeGetTime(); // Записываем текущее время

// Для примера устанавливаем действие атаки
Controller.GetCharacter(0)->Victim = FALSE;
Controller.SetAction(Controller.GetCharacter(0), CHAR_ATTACK);

// Присоединяем оружие к NPC (предмет #0 - меч)
Controller.Equip(Controller.GetCharacter(0), 0, WEAPON, TRUE);

while(1) {
    // Ограничиваем обновления каждые 33 миллисекундами
    while(timeGetTime() < UpdateCounter + 33);

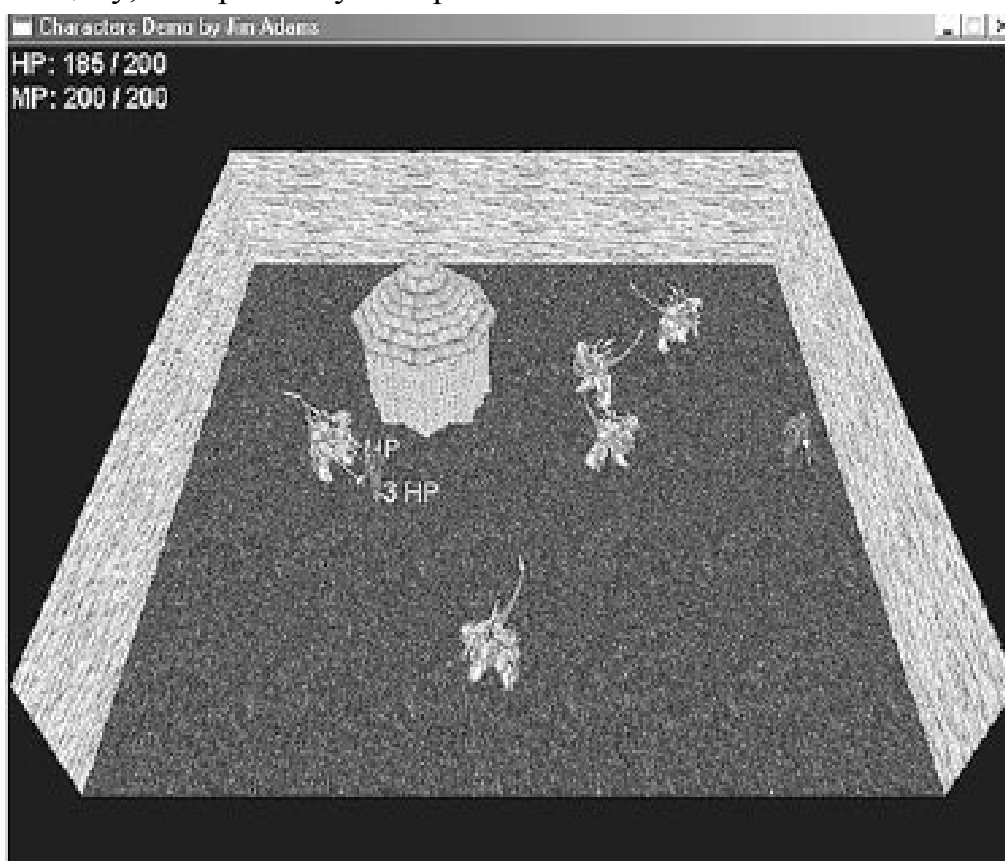
    UpdateCounter = timeGetTime(); // Записываем новое время

    Controller.Update(33);          // Принудительное обновление
                                   // на 33 миллисекунды
    Graphics.Clear();
    if(Graphics.BeginScene() == TRUE) {
        // Обновление анимации персонажа на 33 миллисекунды
        // и визуализация ее на экране
        Controller.Render(33);
    }
    Graphics.Display();
}
```

Этот краткий пример демонстрирует основы использования контроллера. Более сложный пример можете посмотреть в демонстрационной программе Chars.

## Демонстрация персонажей в программе Chars

Весь ваш тяжкий труд окупится при демонстрации контроллеров персонажей и заклинаний, увиденных в этой главе. Пришло время посмотреть демонстрационную программу Chars, включенную на прилагаемый к книге CD-ROM (загляните в \BookCode\Chap12\Chars\). Запустив программу вы увидите сцену, изображенную на рис. 12.17.



*Рис. 12.17. Демонстрационная программа Chars содержит только основы - персонажей, взаимодействующих между собой. В этой программе они ходят, разговаривают и сражаются*

В программе Chars вы управляете PC, используя клавиши управления курсором для его перемещения и разворота. Управление прямолинейно — используйте пробел для взаимодействия с ближайшим персонажем (поговорить с NPC или атаковать врага). Нажатие цифровых клавиш от 1 до 3 приводит к произнесению заклинаний, действующих на ближайшего врага.

Каждый персонаж в игре демонстрирует собственный искусственный интеллект. В разговоре персонаж сообщает, какой тип искусственного интеллекта он использует (за исключением врагов, которые либо стоят, либо преследуют персонаж игрока). Лучше быстро убить врагов, прежде чем они настигнут персонаж игрока.

Все, что есть в программе Chars, было исследовано в этой главе. Скрипты определяют, как персонажи размещаются на карте во время запуска (это задается в стартовом скрипте) и что каждый персонаж делает или говорит, когда к нему обращаются.

Шаблон действий демонстрационной программы, `default.mla`, содержит много действий скрипта, которые напрямую модифицируют тип персонажа, искусственный интеллект, местоположение и направление. Добавление персонажа к миру также просто как использование скриптового действия добавления персонажа, при этом вы соответствующим образом модифицируете атрибуты персонажа.

Что касается главного приложения, класс системного ядра **cApplication** используется для управления потоком выполнения программы; каждое обновление кадра регулируется 33-миллисекундной задержкой, что дает частоту обновления 30 кадров в секунду. В каждом кадре читаются и сохраняются данные клавиатуры, ожидающие использования в функции обновления РС. Фиксированная камера визуализирует действия, с каждым персонажем, полностью анимированным внутри отдельного уровня (и персонажи и уровень представлены сетками).

Код демонстрационной программы хорошо прокомментирован, так что наслаждайтесь его исследованием, и посмотрите, как быстро вы сможете создать бродящие повсюду персонажи в вашем игровом проекте. Конечно же просмотрите скрипты и шаблоны действий скриптов, используя Mad Lib Script Editor, а также определения предметов и персонажей, используя MIL Editor и MCL Editor.

## Заканчиваем с персонажами

Вы можете многое делать с персонажами в вашей игре, и эта глава только коснулась основ. Начать усовершенствование персонажей можно с программирования лучшей системы искусственного интеллекта, лучше обрабатывающей решения, принимаемые персонажами в ходе игры. Добавьте возможности нахождения пути, и внезапно персонажи станут знакомыми с окружением, и будут знать, куда они идут и как туда добраться.

Кроме того, при работе с врагами, имеется базовая структура для определения, какое действие выполнять — исцелить себя, атаковать или направить против оппонента заклинание. Процент вероятности выполнения каждого из этих действий фиксирован. Чтобы улучшить интеллект персонажей попробуйте назначать вероятность использования врагом каждого отдельного действия или магического заклинания.

Всегда помните главное правило — сохраняйте простоту. Хотя класс персонажа, представленный в этой главе, прост, когда вам нужно управлять персонажами он становится мощным дополнением к вашему игровому проекту.

**Программы на CD-ROM**

Каталог \BookCode\Chap12\ содержит следующие программы, демонстрирующие как использовать и редактировать персонажи и заклинания:

**MCLEdit** — обсуждавшийся в этой главе редактор главного списка персонажей. Используйте эту программу для редактирования списка персонажей вашей игры. Местоположение: \BookCode\Chap12\MCLEdit\.

**MSLEdit** — обсуждавшийся в этой главе редактор главного списка заклинаний. Используйте эту программу для редактирования заклинаний и эффектов вашей игры. Местоположение: \BookCode\Chap12\MSLEdit\.

**Chars** — использование этого приложения позволит увидеть как управлять персонажами, включая PC и NPC. В качестве игрока вы управляете действиями PC, используя клавиши управления курсором, пробел и цифровые клавиши 1, 2 и 3 (для произнесения заклинаний). Местоположение: \BookCode\Chap12\Chars\.



# Глава 13

## Работа с картами и уровнями

В главе 8, «Создание трехмерного графического движка», вы узнали, как конструировать и отображать ваши карты и уровни во всем блеске трехмерной графики. Однако, рисование этих довольно маленьких карт — это только начало. Вам также надо разместить в мире персонажи, отметить точки, которые будут запускать определенные действия, разместить двери и другие препятствия, и разработать способ отслеживания где игрок уже был. Ладно, не дрожите; этой главы достаточно, чтобы вы смогли заняться этими задачами.

В этой главе вы узнаете, как делается следующее:

- Заселение карт персонажами.
- Использование триггеров и барьеров.
- Редактирование карт в игре и вне ее.
- Использование автоматической картографии для отслеживания перемещений персонажа.

### Размещение персонажей на карте

В предыдущих примерах из книги я размещал персонажи на карте напрямую, жестко кодируя их. Однако помните, что жестко кодировать игровые данные — дурной тон. Вы должны обеспечить максимум гибкости для проектирования ваших карт, и в это входит размещение персонажей на уровне.

Два метода размещения персонажей на карте, обеспечивающих желаемую гибкость, включают список персонажей карты и скрипты.

### Список персонажей карты

В ряде глав, например, в главе 10, «Реализация скриптов», я использовал внешние файлы данных, которые хранили набор чисел и текстов. Эти файлы данных загружались и преобразовывались в некую полезную информацию для загружающего данные движка. Например, шаблоны действий, содержат текст действия и данные элементов для каждого действия, и все в одном, простом для чтения и редактирования файле.

Придерживаясь простой природы использования текстовых файлов данных, вы можете создать список персонажей, размещаемых на карте, когда карта загружена. Поскольку для размещения игроков на карте применяется только набор координат и направление взгляда, этот файл данных может выглядеть следующим образом:

```
0 100.0 0.0 450.0 0.0
21 0.0 0.0 -82.0 1.57
18 640.0 10.0 0.0 3.14
```

На первый взгляд это только три строки обычных чисел, но тренированный взгляд видит, что каждое число представляет что-нибудь особенное. Начиная с первого числа в каждой строке вы получаете следующее:

- Тип персонажа (например, 0 — главный персонаж, 21 — огр, 18 — ребенок).
- Координаты X, Y и Z.
- Угол (в радианах), в котором обращен персонаж.

Теперь, зная что означает каждое число, вы видите, что я определил три персонажа, разместил их на карте в соответствующих местах и обратил в заданном направлении. Эти данные компактны, просты для редактирования, и могут быть быстро загружены и обработаны.

### ***Загрузка списка персонажей карты***

Чтобы обработать файл данных, как было только что описано, вам необходимы только две функции (которые вы уже видели в главе 10). Вот эти функции:

```
long GetNextLong(FILE *fp)
{
    char Buf[1024];
    long Pos = 0;
    int c;

    // Читаем, пока нет EOF или EOL
    while(1) {
        if((c = fgetc(fp)) == EOF)
            break;
        if(c == 0x0a || (c == ' ' && Pos))
            break;
        if((c >= '0' && c <= '9') || c == '.' || c == '-')
            Buf[Pos++] = c;
    }

    if(!Pos)
        return -1;

    Buf[Pos] = 0;

    return atol(Buf);
}
```

```
float GetNextFloat(FILE *fp)
{
    char Buf[1024];
    long Pos = 0;
    int c;

    // Читаем, пока нет EOF или EOL
    while(1) {
        if((c = fgetc(fp)) == EOF)
            break;
        if(c == 0x0a || (c == ' ' && Pos))
            break;
        if((c >= '0' && c <= '9') || c == '.' || c == '-')
            Buf[Pos++] = c;
    }

    Buf[Pos] = 0;

    return (float)atof(Buf);
}
```

Обе функции получают в аргументе указатель на файл (**fp**) и возвращают следующее число типа **long** или **float**, найденное в указанном файле. Вы организуете данные списка персонажей карты таким образом, что первое число (тип персонажа) будет типа **long**, а все остальные числа будут **float**.

Используя **GetNextLong** и **GetNextFloat** вы можете выполнить разбор всего списка персонажей карты следующим образом:

```
// fp = указатель на открытый файл с данными персонажей карты
long Type; // Тип загружаемого персонажа
float XPos, YPos, ZPos, Direction;
long NumCharactersLoaded = 0; // Кол-во загруженных персонажей

while(1) {
    // Прерываемся, если больше нет персонажей для обработки
    if((Type = GetNextLong(fp)) == -1)
        break;

    // Получаем координаты и угол
    XPos = GetNextFloat(fp);
    YPos = GetNextFloat(fp);
    ZPos = GetNextFloat(fp);
    Direction = GetNextFloat(fp);

    // Делаем что-нибудь с данными - вставляем персонаж

    NumCharactersLoaded++; // Увеличиваем кол-во загруженных персонажей
}
// Завершили загрузку, в NumCharactersLoaded
// кол-во загруженных персонажей
```

### ***Использование списка персонажей карты в вашей игре***

Использование списка персонажей карты это быстрый способ разместить персонажи на карте. Когда вы загрузили карту в память, загрузите соответствующий список персонажей карты и вставьте персонажи. Хотя этот



метод добавления персонажей к карте выглядит удовлетворительно, иногда вам может потребоваться больше гибкости, и здесь в игру вступает использование скриптов.

## Скриптовое размещение

Как я говорил раньше в этом разделе, вы можете также для размещения персонажей на карте использовать скрипты. Скрипты дают вам больший контроль над тем где и когда будут размещены персонажи на карте, чем использование прямого размещения.

Представьте, например, что в своей игре вы отслеживаете время. Ночью все горожане сидят дома, поэтому рынок пуст. На торговых местах стоят стражники, так что вашей игре необходимо знать, какие персонажи размещать — торговцев или стражников.

Вы можете заметить, что я подошел к этому моменту с пустыми руками, не определив класс для загрузки размещения персонажей на карте. Как персонажи будут знать, где им разместиться на ваших картах?

Как вы могли предположить, можно использовать систему Mad Lib Script, представленную в главе 10, создав и включив шаблон действий и обработчик скриптов в ваш проект — тогда обработчик скрипта берет на себя задачу добавления персонажей на карту на основании скриптов, которые вы напишете для вашей игры.

Пример шаблона действий, который добавляет персонажи к карте, может выглядеть так:

```
"Add character # ~ to map at coordinates ~, ~, ~"
INT 0 65535
FLOAT -5000 5000
FLOAT -5000 5000
FLOAT -5000 5000
```

Определив это единственное действие вы можете конструировать небольшие скрипты, размещающие персонажи (с уникальным идентификационным номером) на карте, основываясь на предоставляемых вами координатах X, Y и Z. Быстрое, чистое и верное по сути использование скриптов определенно является тем путем, по которому следует идти, и именно этот метод я использую для добавления персонажей на карту в законченной ролевой игре из главы 16, «Объединяем все вместе в законченную игру».

## Использование триггеров карты

Причина и эффект — вот два слова, которые говорят все. В вашем мире ничего не происходит, пока вы явно не скажете об этом. Большинство событий в вашей игре происходят когда персонаж берет предмет, проходит мимо заданной области или даже пытается поговорить с другим персонажем. Эти события называются *триггерами* (*trigger*), и как только триггер

срабатывает, будьте уверены, что последует ряд эффектов. Эти эффекты обычно принимают форму обрабатываемого скрипта.

Проблема здесь не в том, как иметь дело со скриптами, а как определить, что их надо запустить. Программировать триггеры для таких вещей, как взятие предметов, достаточно легко; просто назначьте номер в описании предмета и обрабатывайте соответствующий скрипт, если предмет взят. То же самое верно и для разговора с персонажами.

С картами дело обстоит по-другому. Карты бывают самых разных форм и размеров и задачей здесь является определить момент, когда персонаж касается определенной области карты. Ладно, я пошутил, не такая уж это и проблема. Трюк заключается в пометке областей карты с помощью геометрических тел, для которых можно быстро определить, вступил ли персонаж внутрь них.

Геометрические тела, которые вы будете использовать, это сферы, параллелепипеды, цилиндры и треугольные призмы. Взгляните на каждое из них и как они работают внутри общей схемы триггеров.

## Сферические триггеры

Вы определяете сферический триггер (рис. 13.1) набором координат и радиусом. У сферического триггера есть два уникальных преимущества:

- Сферы замечательно подходят для определения в качестве триггеров больших областей карты, используя для описания местоположения сферы только координаты центра и радиус.
- Сферический триггер это один из наиболее быстрых способов проверить столкновение персонажа с триггером в движке триггеров карты.

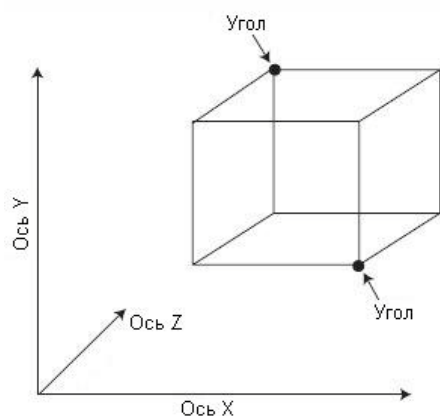


*Рис. 13.1. Вы можете разместить сферический триггер в трехмерном пространстве, используя трио координат и радиус сферы*

## Кубический триггер

Кубический триггер использует для работы своей магии ограничивающий параллелепипед. Кубический триггер является самым быстрым при проверке столкновения персонажа с триггером, но у него есть недостаток — стороны кубического триггера должны быть параллельны мировым осям (вы не можете вращать параллелепипед, чтобы приспособить его к вашим

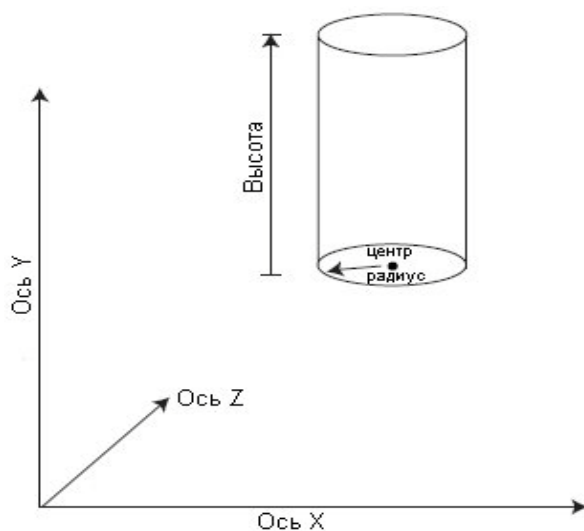
потребностям). Определяется кубический триггер путем задания координат двух противоположных углов, как показано на рис. 13.2.



**Рис. 13.2.** Кубический триггер определяется координатами противоположных углов

## Цилиндрический триггер

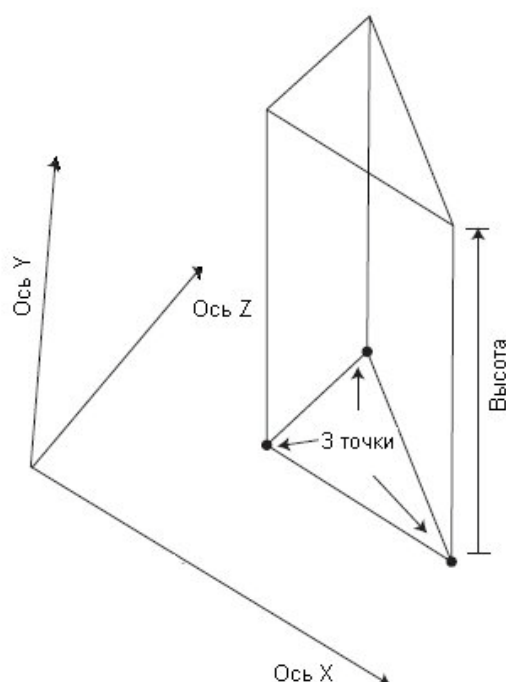
Цилиндрический триггер во многом похож на сферический, за исключением того, что в цилиндрическом триггере вы можете ограничить высоту охватываемой области (в отличие от сферического триггера, который простирается в высоту на длину радиуса). Цилиндрические триггеры наиболее эффективны при использовании для круглых областей, в которых вы хотите удержать высоту области срабатывания от распространения выше или ниже заданного уровня. Пример цилиндрического триггера показан на рис. 13.3.



**Рис. 13.3.** Цилиндрический триггер определяется координатами центра нижней грани, радиусом и высотой

## Треугольный триггер

Треугольный триггер подобен полигону в том, что оба определяются тремя точками; однако в треугольном триггере три точки определяются только их координатами  $X$  и  $Z$ . Это делает треугольник двумерным. Чтобы треугольник работал в трехмерном мире, вы должны назначить единую координату  $Y$  для размещения всех трех точек треугольника, а также высоту, на которую треугольная область распространяется вверх. Можно думать о треугольном триггере, как о трехстороннем ящике, что показано на рис. 13.4.



**Рис. 13.4.** Используя плоский двухмерный треугольник, простирающийся вверх от своего местоположения в мире, треугольные триггеры являются наиболее универсальной формой, которую можно использовать для триггеров нижней грани, радиусом и высотой

## Срабатывание триггеров

После того, как вы разместите формы триггеров на карте, остается простой вопрос определения того, какого триггера коснулся персонаж. У каждого триггера есть особый способ обнаружения этих столкновений персонажа и триггера. Сфера использует проверку расстояния, куб использует вычисления ограничивающего параллелепипеда, а цилиндр использует ограничения и проверку расстояния, треугольные триггеры используют проверку ограничений, а также убеждаются, что интересующая точка находится внутри треугольника.

### ПРИМЕЧАНИЕ

Вы увидите, как выполняется каждая проверка столкновений в последующем разделе «Создание класса триггера».

Что делать, когда вы обнаружите, что триггер сработал? Поскольку каждому триггеру назначен идентификационный номер, вы можете использовать этот номер, чтобы определить, какое действие выполнять. Вы можете исполнить соответствующий скрипт или выполнить другую жестко закодированную функцию. Фактически, в последующем разделе «Использование триггеров» вы увидите, как с пользой применять триггеры.

## Создание класса триггера

Придерживаясь техники объектно-ориентированного программирования, создадим класс, который будет поддерживать список триггеров и определять какого из них (если такой есть) коснулся персонаж. Класс использует структуру для хранения информации о каждом триггере — координаты, тип и т.д. Каждому триггеру также назначен идентификационный номер,

который используется для ссылки на него. Весь набор триггеров хранится как связанный список структур.

Класс **cTrigger** может загружать и сохранять файл триггеров, что упрощает редактирование списка триггеров. Этот файл является текстовым и его просто читать и редактировать. Для каждого триггера на карте используется отдельная строка текста, записанная в следующем порядке: идентификационный номер, тип триггера (0 — сферический, 1 — кубический, 2 — цилиндрический, 3 — треугольный) и состояние триггера по умолчанию (будет ли триггер взведен после загрузки). Значение 0 означает, что триггер отключен, а значение 1 означает, что триггер включен.

В описание триггера входит еще несколько значений, зависящих от типа описываемого триггера. Сфере требуются координаты X, Y и Z и радиус, как показано ниже:

```
ID 0 ENABLED X Y Z RADIUS
```

Для параллелепипеда необходимы координаты противоположных углов:

```
ID 1 ENABLED X1 Y1 Z1 X2 Y2 Z2
```

Цилиндр вы определяете, задавая координаты центра нижней грани, плюс радиус и высоту:

```
ID 2 ENABLED X Y Z RADIUS HEIGHT
```

И, наконец, треугольник вы определяете по координатам X и Z трех углов, упорядочивая их по часовой стрелке, когда на треугольник смотрят по оси Y (точно так же, как треугольные грани определялись в главе 2, «Рисование с DirectX Graphics»). Завершают определение триггера общая координата Y всех трех точек и его высота:

```
ID 3 ENABLED X1 Z1 X2 Z2 X3 Z3 Y HEIGHT
```

Через минуту я вернусь к файлу данных триггеров. А сейчас взгляните на определение класса триггера. Класс (его заголовок объявлен в файле `Trigger.h`, а полный исходный код класса находится в файле `Trigger.cpp`) начинается с перечисления, в котором определены типы форм триггеров, которые вы можете использовать:

```
// Перечисление типов триггеров
enum TriggerTypes { Trigger_Sphere = 0, Trigger_Box,
                   Trigger_Cylinder, Trigger_Triangle };
```

Каждому определяемому триггеру требуется структура, содержащая относящуюся к триггеру информацию — местоположение триггера, флаг активности и уникальный идентификационный номер. Каждый тип триггера использует набор координат для определения своего местоположения на карте, а также дополнительные данные, чтобы определить радиус триггера, координаты противоположных углов и т.д. Структура, содержащая информацию о каждом созданном триггере, выглядит так:

```

typedef struct sTrigger {
    long Type;           // Сфера, куб и т.д.
    long ID;             // ID триггера
    BOOL Enabled;        // Флаг активности

    float x1, y1, z1;    // Координаты 1
    float x2, y2, z2;    // Координаты 2
    float x3, z3;        // Координаты 3
    float Radius;        // Радиус границ

    sTrigger *Prev, *Next; // Связанный список триггеров

    sTrigger() { Prev = Next = NULL; }
    ~sTrigger() { delete Next; Next = NULL }
} sTrigger;

```

Заметьте, что структура **sTrigger** содержит набор указателей связанного списка, а также содержит конструктор и деструктор, которые очищают указатели связанного списка и освобождают связанный список, соответственно.

Для работы со структурой **sTrigger** вы используете класс триггера, который управляет связанным списком триггеров и позволяет вам сохранять и загружать список этих триггеров. Взгляните на объявление класса триггера:

```

class cTrigger
{
private:
    long m_NumTriggers; // Кол-во триггеров в связанном списке
    sTrigger *m_TriggerParent; // Родитель связанного списка

    long GetNextLong(FILE *fp); // Получаем следующее long
    float GetNextFloat(FILE *fp); // Получаем следующее float

    // Функция, добавляющая триггер к связанному списку
    sTrigger *AddTrigger(long Type, long ID, BOOL Enabled);

public:
    cTrigger();
    ~cTrigger();

    // Функции для загрузки/сохранения списка триггеров
    BOOL Load(char *Filename);
    BOOL Save(char *Filename);

    // Функции для добавления определенного триггера к списку
    BOOL AddSphere(long ID, BOOL Enabled,
                   float XPos, float YPos, float ZPos,
                   float Radius);

    BOOL AddBox(long ID, BOOL Enabled,
                float XMin, float YMin, float ZMin,
                float XMax, float YMax, float ZMax);

    BOOL AddCylinder(long ID, BOOL Enabled,
                     float XPos, float YPos, float ZPos,
                     float Radius, float Height);

```

```
    BOOL AddTriangle(long ID, BOOL Enabled,
                    float x1, float z1,
                    float x2, float z2,
                    float x3, float z3,
                    float YPos, float Height);

    // Удаление триггера с указанным ID
    BOOL Remove(long ID);

    // Освобождение всех триггеров
    BOOL Free();

    // Поиск первого триггера в заданном месте
    // (возвращает 0 если нет)
    long GetTrigger(float XPos, float YPos, float ZPos);

    // Получение состояния триггера с указанным ID
    BOOL GetEnableState(long ID);

    // Включение/отключение триггера с указанным ID
    BOOL Enable(long ID, BOOL Enable);

    // Возвращает количество триггеров
    // и родителя связанного списка
    long GetNumTriggers();
    sTrigger *GetParentTrigger();
};
```

Большинство функций имеют дело только со связанным списком структур **sTrigger** — добавляют структуру, удаляют структуру, ищут структуру и модифицируют ее и т.д. Чтобы яснее понимать, что будет дальше, уделите минуту или две на обзор следующих разделов, где показан код для каждой функции.

### ***cTrigger::cTrigger* и *cTrigger::~~cTrigger***

Как и каждый класс C++, **cTrigger** имеет конструктор и деструктор, которые инициализируют и освобождают содержащиеся внутри класса данные. Единственными данными, которые отслеживаются классом триггера и не содержатся в связанном списке, являются текущее количество триггеров в связанном списке и указатель на этот связанный список. Конструктор и деструктор гарантируют, что класс подготовит для использования эти две переменные и данные класса будут освобождены при удалении объекта (вызовом функции **Free**), что показано ниже:

```
cTrigger::cTrigger() {
    m_NumTriggers = 0;
    m_TriggerParent = NULL;
}

cTrigger::~~cTrigger()
{
    Free();
}
```

## ***cTrigger::Load* и *cTrigger::Save***

Вы обычно разрабатываете карту с набором триггеров в соответствующих местах. Загрузка списка этих триггеров — основной приоритет класса триггера. Когда список триггеров создан или загружен, у вас есть также возможность сохранить этот список триггеров (например, для сохранения состояния игры).

Функция **Load** открывает текстовый файл и читает строки текста, определяющие тип, идентификатор, местоположение и специальные свойства каждого триггера (как было описано в предыдущем разделе «Создание класса триггера»). Когда достигнут конец файла, функция **Load** возвращает управление. Взгляните на код функции **Load**, чтобы увидеть, о чем я говорю:

```
BOOL cTrigger::Load(char *Filename)
{
    FILE *fp;
    long Type, ID;
    BOOL Enabled;
    float x1, y1, z1, x2, y2, z2, x3, z3, Radius;

    Free(); // Удаляем все текущие триггеры

    // Открываем файл
    if((fp=fopen(Filename, "rb")) == NULL)
        return FALSE;
```

Сейчас файл данных триггеров открыт и готов к началу чтения списка определений триггеров. Помните, что для каждого триггера в текстовой строке установлен следующий порядок: идентификационный номер триггера, тип (0 — сфера, 1 — куб и т.д.), состояние триггера по умолчанию (0 — триггер отключен, 1 — включен) и специфические данные, зависящие от типа триггера. Держите в уме этот порядок, продолжая чтение:

```
// Начинаем чтение, пока не достигнем EOF
while(1) {
    // Получаем ID триггера
    if((ID = GetNextLong(fp)) == -1)
        break;

    Type = GetNextLong(fp); // Получаем тип

    // Получаем состояние включения
    Enabled = (GetNextLong(fp)) ? TRUE : FALSE;

    // Дальше читаем в зависимости от типа
    switch(Type) {
        case Trigger_Sphere: // Загрузка сферы
            x1 = GetNextFloat(fp); y1 = GetNextFloat(fp);
            z1 = GetNextFloat(fp); Radius = GetNextFloat(fp);
            AddSphere(ID, Enabled, x1, y1, z1, Radius);
            break;
        case Trigger_Box: // Загрузка куба
            x1 = GetNextFloat(fp); y1 = GetNextFloat(fp);
            z1 = GetNextFloat(fp); x2 = GetNextFloat(fp);
            y2 = GetNextFloat(fp); z2 = GetNextFloat(fp);
```



```
        AddBox(ID, Enabled, x1, y1, z1, x2, y2, z2);
        break;
    case Trigger_Cylinder: // Загрузка цилиндра
        x1 = GetNextFloat(fp); y1 = GetNextFloat(fp);
        z1 = GetNextFloat(fp); Radius = GetNextFloat(fp);
        y2 = GetNextFloat(fp);
        AddCylinder(ID, Enabled, x1, y1, z1, Radius, y2);
        break;
    case Trigger_Triangle: // Загрузка треугольника
        x1 = GetNextFloat(fp); z1 = GetNextFloat(fp);
        x2 = GetNextFloat(fp); z2 = GetNextFloat(fp);
        x3 = GetNextFloat(fp); z3 = GetNextFloat(fp);
        y1 = GetNextFloat(fp); y2 = GetNextFloat(fp);
        AddTriangle(ID, Enabled, x1, z1, x2, z2,
                    x3, z3, y1, y2);
        break;
    default: fclose(fp); // Произошла какая-то ошибка
        Free();
        return FALSE;
    }
}

// Закрываем файл и возвращаем результаты
fclose(fp);

return (m_NumTriggers) ? TRUE : FALSE;
}
```

После чтения идентификационного номера, типа и флага активности каждого триггера, отдельный кодовый блок **switch...case** заботится о загрузке данных триггеров каждого типа. После чтения данных триггера вызывается отдельная функция (зависящая от типа триггера) для вставки триггера в связанный список. Эти функции **AddSphere**, **AddBox**, **AddCylinder** и **AddTriangle**.

Оставив позади функцию **Load**, вы видите функцию **Save**, которая сканирует связанный список триггеров и записывает данные каждого триггера в файл, используя тот же самый формат для каждой описывающей триггер строки текста. Взгляните:

```
BOOL cTrigger::Save(char *Filename)
{
    FILE *fp;
    sTrigger *TriggerPtr;

    // Контроль ошибок
    if(!m_NumTriggers)
        return FALSE;
    if((TriggerPtr = m_TriggerParent) == NULL)
        return FALSE;

    // Открываем файл
    if((fp=fopen(Filename, "wb"))== NULL)
        return FALSE;

    // Записываем все триггеры из связанного списка
    while(TriggerPtr != NULL) {
        // Записываем ID, тип и флаг активности
        fprintf(fp, "%lu ", TriggerPtr->ID);
```

```

fprintf(fp, "%lu ", TriggerPtr->Type);
fprintf(fp, "%lu ", (TriggerPtr->Enabled) ? 1 : 0);

// Записываем оставшиеся данные в зависимости от типа
switch(TriggerPtr->Type) {
    case Trigger_Sphere: // Записываем сферу
        fprintf(fp, "%lf %lf %lf %lf %lf\r\n",
            TriggerPtr->x1, TriggerPtr->y1, TriggerPtr->z1,
            TriggerPtr->Radius);
        break;
    case Trigger_Box: // Записываем куб
        fprintf(fp, "%lf %lf %lf %lf %lf %lf %lf\r\n",
            TriggerPtr->x1, TriggerPtr->y1, TriggerPtr->z1,
            TriggerPtr->x2, TriggerPtr->y2, TriggerPtr->z2);
        break;
    case Trigger_Cylinder: // Записываем цилиндр
        fprintf(fp, "%lf %lf %lf %lf %lf %lf\r\n",
            TriggerPtr->x1, TriggerPtr->y1, TriggerPtr->z1,
            TriggerPtr->Radius, TriggerPtr->y2);
        break;
    case Trigger_Triangle: // Записываем треугольник
        fprintf(fp, "%lf %lf %lf %lf %lf %lf %lf %lf %lf\r\n",
            TriggerPtr->x1, TriggerPtr->z1,
            TriggerPtr->x2, TriggerPtr->z2,
            TriggerPtr->x3, TriggerPtr->z3,
            TriggerPtr->y1, TriggerPtr->y2);
        break;
}
}

// Закрываем файл и сообщаем об успехе
fclose(fp);

return TRUE;
}

```

### ***cTrigger::AddTrigger***

**AddTrigger** — это сердце всех других функций, которые добавляют триггеры. Эта функция выделяет память под структуру **sTrigger**, устанавливает тип, идентификационный номер и флаг активности, после чего вставляет структуру в связанный список триггеров. Как только ваша программа выделит память, используя функцию **AddTrigger**, можно заполнить возвращенную структуру **sTrigger** координатами, радиусом, высотой и любой другой информацией, необходимой для определения триггера.

Держа в уме, что функция **AddTrigger** только выделяет память под структуру **sTrigger** и заполняет ее минимальным набором упомянутых данных, взгляните на код:

```

sTrigger *cTrigger::AddTrigger(long Type, long ID, BOOL Enabled)
{
    // Выделяем новую структуру триггера и включаем ее в список
    sTrigger *Trigger = new sTrigger();

    Trigger->Prev = NULL;
    if((Trigger->Next = m_TriggerParent) != NULL)

```

```
        m_TriggerParent->Prev = Trigger;
    m_TriggerParent = Trigger;

    // Устанавливаем тип, ID и флаг активности триггера
    Trigger->Type = Type;
    Trigger->ID = ID;
    Trigger->Enabled = Enabled;

    m_NumTriggers++; // Увеличиваем счетчик триггеров

    return Trigger; // Возвращаем указатель на структуру
}
```

### ***cTrigger::AddSphere, cTrigger::AddBox, cTrigger::AddCylinder и cTrigger::AddTriangle***

Эта группа функций добавляет триггеры заданного типа к связанному списку триггеров. У каждой функции свой собственный список аргументов, используемых для создания (вы можете посмотреть предшествующие каждой функции комментарии, чтобы увидеть, что делает каждый аргумент). Независимо от типа триггера каждая функция сперва вызывает функцию **AddTrigger**, чтобы получить структуру **sTrigger** с которой она будет работать.

Давайте начнем с функции **AddSphere**, которая получает, помимо идентификационного номера триггера и состояния активности (так здесь поступает каждая из четырех функций), еще радиус сферы и координаты *x*, *y* и *z* ее центра:

```
BOOL cTrigger::AddSphere(long ID, BOOL Enabled,
                        float XPos, float YPos, float ZPos,
                        float Radius)
{
    // Создаем новую структуру триггера и включаем ее в список
    sTrigger *Trigger = AddTrigger(Trigger_Sphere, ID, Enabled);

    // Устанавливаем данные триггера
    Trigger->x1 = XPos;
    Trigger->y1 = YPos;
    Trigger->z1 = ZPos;
    Trigger->Radius = Radius * Radius;

    return TRUE;
}
```

---

#### **ПРИМЕЧАНИЕ**

Все функции, получающие в качестве аргумента радиус сохраняют в структуре квадрат указанного значения. В дальнейшем это ускорит проверку дистанции. Как класс триггера ускоряет проверку дистанции? Стандартная проверка дистанции использует вызов **sqrt** для вычисления правильного расстояния. Отказ от **sqrt** ускоряет работу движка, но в этом случае вы должны использовать квадраты значений для сопоставления дистанции.

---

Говоря кратко и по сути, функция **AddSphere** вызывает функцию **AddTrigger** для выделения памяти под структуру **sTrigger** и включения ее в связанный список. После создания экземпляра структуры **sTrigger** заполняется координатами и радиусом сферического триггера.

**AddBox**, **AddCylinder** и **AddTriangle** действуют тем же образом, что и функция **AddSphere**. Функция **AddBox** получает идентификационный номер и состояние активности, а также координаты двух противоположных углов параллелепипеда:

```

BOOL cTrigger::AddBox(long ID, BOOL Enabled,
                      float XMin, float YMin, float ZMin,
                      float XMax, float YMax, float ZMax)
{
    // Создаем новую структуру триггера и включаем ее в список
    sTrigger *Trigger = AddTrigger(Trigger_Box, ID, Enabled);

    // Устанавливаем данные триггера
    // (упорядочивая минимальные и максимальные значения)
    Trigger->x1 = min(XMin, XMax);
    Trigger->y1 = min(YMin, YMax);
    Trigger->z1 = min(ZMin, ZMax);
    Trigger->x2 = max(XMin, XMax);
    Trigger->y2 = max(YMin, YMax);
    Trigger->z2 = max(ZMin, ZMax);

    return TRUE;
}

```

Функция **AddCylinder** использует для триггера координаты центра нижней грани цилиндра, радиус и высоту. Взгляните на код **AddCylinder**:

```

BOOL cTrigger::AddCylinder(long ID, BOOL Enabled,
                           float XPos, float YPos, float ZPos,
                           float Radius, float Height)
{
    // Создаем новую структуру триггера и включаем ее в список
    sTrigger *Trigger = AddTrigger(Trigger_Cylinder, ID, Enabled);

    // Устанавливаем данные триггера
    Trigger->x1 = XPos;
    Trigger->y1 = YPos;
    Trigger->z1 = ZPos;
    Trigger->Radius = Radius * Radius;
    Trigger->y2 = Height;

    return TRUE;
}

```

Завершает набор **AddTriangle**, которая получает три пары координат X и Z, определяющих каждый из трех углов треугольника. Координата Y является общей для всех трех углов, также как и следующая за ней высота фигуры треугольного триггера. Теперь следующий фрагмент кода должен быть ясен:

```
BOOL cTrigger::AddTriangle(long ID, BOOL Enabled,
                           float x1, float z1,
                           float x2, float z2,
                           float x3, float z3,
                           float YPos, float Height)
{
    // Создаем новую структуру триггера и включаем ее в список
    sTrigger *Trigger = AddTrigger(Trigger_Triangle, ID, Enabled);

    // Устанавливаем данные триггера
    Trigger->x1 = x1;
    Trigger->z1 = z1;
    Trigger->x2 = x2;
    Trigger->z2 = z2;
    Trigger->x3 = x3;
    Trigger->z3 = z3;
    Trigger->y1 = YPos;
    Trigger->y2 = Height;

    return TRUE;
}
```

### ***cTrigger::Remove* и *cTrigger::Free***

Эти две функции удаляют триггеры из связанного списка, позволяя указать идентификационный номер удаляемого триггера в функции **Remove**, или позволяя классу удалить все триггеры из списка, используя функцию **Free**.

Функция **Remove** сканирует весь связанный список и для каждого триггера, идентификационный номер которого совпадает с тем, который передан функции **Remove** в аргументе **ID**, функция **Remove** удаляет структуру из связанного списка и освобождает занятую структурой память:

```
BOOL cTrigger::Remove(long ID)
{
    sTrigger *TriggerPtr, *NextTrigger;
    long Count = 0;

    // Сканируем список триггеров
    if((TriggerPtr = m_TriggerParent) != NULL) {
        while(TriggerPtr != NULL) {
```

Здесь начинается сканирование связанного списка структур **sTrigger**. Вы сохраняете указатель на следующую структуру в связанном списке и проверяете текущую структуру **sTrigger** на соответствие с идентификационным номером, который должен быть удален:

```
        // Запоминаем следующий элемент
        NextTrigger = TriggerPtr->Next;

        // Совпадает?
        if(TriggerPtr->ID == ID) {
```

Когда определено, что структура должна быть удалена, следующий код меняет указатели связанного списка и освобождает занятую структурой память:

```

// Удаляем из списка
if (TriggerPtr->Prev != NULL)
    TriggerPtr->Prev->Next = TriggerPtr->Next;
else
    m_TriggerParent = TriggerPtr->Next;
if (TriggerPtr->Next != NULL)
    TriggerPtr->Next->Prev = TriggerPtr->Prev;
if (TriggerPtr->Prev == NULL && TriggerPtr->Next == NULL)
    m_TriggerParent = NULL;

// Освобождаем память
TriggerPtr->Prev = TriggerPtr->Next = NULL;
delete TriggerPtr;

```

Теперь уменьшается количество триггеров, хранящихся в связанном списке, и цикл сканирования структур для удаления продолжается, пока не будут перебраны все структуры:

```

// Уменьшаем количество триггеров
// и увеличиваем счетчик удаленных
m_NumTriggers--;
Count++;
}

// Переходим к следующему триггеру
TriggerPtr = NextTrigger;
}

}

// Возвращаем TRUE если что-то нашли и удалили
return (Count) ? TRUE : FALSE;
}

```

В то время, как функция **Remove** удаляет триггеры согласно их идентификационным номерам, функция **Free** может пропустить всю эту суету и одним махом удаляет весь связанный список, используя следующий код:

```

BOOL cTrigger::Free()
{
    delete m_TriggerParent;
    m_TriggerParent = NULL;
    m_NumTriggers = 0;

    return TRUE;
}

```

### ***cTrigger::GetTrigger***

**GetTrigger** — это функция класса триггера, вызываемая каждый раз, когда персонаж игрока перемещается. **GetTrigger** получает координаты проверяемого вами персонажа и возвращает идентификационный номер первого триггера, обнаруженного в указанном месте (если он есть). Если в указанном месте триггеров не обнаружено, **GetTrigger** возвращает ноль.

**GetTrigger** выполняет большую работу, но все не так уж и сложно. Сканируется связанный список триггеров и каждый рассматриваемый

триггер проверяется, чтобы увидеть, занимает ли он и указанные координаты одно и то же место на карте. Если да, возвращается идентификационный номер триггера.

---

**ВНИМАНИЕ!**

Никогда не назначайте триггеру нулевой идентификатор, поскольку класс триггера использует ноль, чтобы сообщать при вызове функции **GetTrigger** о том, что никакой триггер не найден.

---

```
long cTrigger::GetTrigger(float XPos, float YPos, float ZPos)
{
    float XDiff, YDiff, ZDiff, Dist;
    D3DXVECTOR2 vecNorm;
    sTrigger *Trigger;

    // Сканируем список триггеров
    if((Trigger = m_TriggerParent) != NULL) {
        while(Trigger != NULL) {
            // Проверяем только активные
            if(Trigger->Enabled == TRUE) {
```

Теперь вы проверяете активный триггер, чтобы увидеть, пересекается ли он с координатами, переданными в аргументах **XPos**, **YPos** и **ZPos** функции **GetTrigger**. У каждого триггера есть специальный способ определения того, находятся ли указанные координаты внутри пространства триггера, и, используя инструкцию **switch**, следующий код выбирает способ выполнения проверки пересечения:

```
        // Проверка в зависимости от типа
        switch(Trigger->Type) {
            case Trigger_Sphere:
```

Для сферы вы применяете проверку расстояния. Если расстояние до точки с заданными координатами меньше или равно радиусу сферы, персонаж коснулся триггера:

```
                // Проверка расстояния для сферы
                // (используется радиус)
                XDiff = (float)fabs(Trigger->x1 - XPos);
                YDiff = (float)fabs(Trigger->y1 - YPos);
                ZDiff = (float)fabs(Trigger->z1 - ZPos);
                Dist = XDiff*XDiff + YDiff*YDiff + ZDiff*ZDiff;

                if(Dist <= Trigger->Radius)
                    return Trigger->ID;
                break;
            case Trigger_Box:
```

Кубический триггер использует обычный ограничивающий параллелепипед для сравнения координат противоположных углов с проверяемыми координатами, чтобы увидеть пересекаются ли они:

```
                // Проверка нахождения внутри параллелепипеда
                if(XPos >= Trigger->x1 && XPos <= Trigger->x2) {
                    if(YPos >= Trigger->y1 && YPos <= Trigger->y2) {
                        if(ZPos >= Trigger->z1 &&
                            ZPos <= Trigger->z2)
```

```

        return Trigger->ID;
    }
}
break;
case Trigger_Cylinder:

```

Цилиндрический триггер использует смесь сферы и ограничивающего параллелепипеда:

```

// Сначала проверяем ограничение высоты
if(YPos >= Trigger->y1 &&
    YPos <= Trigger->y1 + Trigger->y2) {
    // Проверка расстояния от цилиндра
    XDiff = (float)fabs(Trigger->x1 - XPos);
    YDiff = (float)fabs(Trigger->y1 - YPos);
    ZDiff = (float)fabs(Trigger->z1 - ZPos);

    Dist = XDiff*XDiff + YDiff*YDiff + ZDiff*ZDiff;

    if(Dist <= Trigger->Radius)
        return Trigger->ID;
}
break;
case Trigger_Triangle:

```

Показанный здесь код треугольного триггера, проверяет находятся ли указанные координаты перед всеми тремя гранями треугольника, используя скалярное произведение. Для каждой грани треугольника вычисляется скалярное произведение и затем проверяется, чтобы увидеть, находятся ли рассматриваемые координаты внутри или вне треугольника.

Вы можете думать о скалярном произведении как о расстоянии рассматриваемых координат от грани треугольника. Отрицательное расстояние означает, что проверяемые координаты находятся вне треугольника, в то время как положительное расстояние означает, что координаты внутри треугольника.

Если все три проверки скалярного произведения дают положительные значения, проверяемые координаты должны находиться внутри треугольника. Вы используете одну дополнительную проверку, чтобы определить, попадают ли координаты в заданное ограничение высоты, указанное в структуре **sTrigger**:

```

// Сперва проверяем ограничение высоты
if(YPos >= Trigger->y1 &&
    YPos <= Trigger->y1 + Trigger->y2) {

    // Проверяем, что точка перед всеми линиями
    // от x1,z1 к x2,z2
    D3DXVec2Normalize(&vecNorm,
        &D3DXVECTOR2(Trigger->z2 - Trigger->z1,
            Trigger->x1 - Trigger->x2));
    if(D3DXVec2Dot(&D3DXVECTOR2(XPos-Trigger->x1,
        ZPos-Trigger->z1),
        &vecNorm) < 0)
        break;
}

```



```
// от x2,z2 к x3,z3
D3DXVec2Normalize(&vecNorm,
    &D3DXVECTOR2 (Trigger->z3 - Trigger->z2,
        Trigger->x2 - Trigger->x3));
if (D3DXVec2Dot (&D3DXVECTOR2 (XPos-Trigger->x2,
    ZPos-Trigger->z2),
    &vecNorm) < 0)
    break;

// от x3,z3 к x1,z1
D3DXVec2Normalize(&vecNorm,
    &D3DXVECTOR2 (Trigger->z1 - Trigger->z3,
        Trigger->x3 - Trigger->x1));
if (D3DXVec2Dot (&D3DXVECTOR2 (XPos-Trigger->x3,
    ZPos-Trigger->z3),
    &vecNorm) < 0)
    break;

    return Trigger->ID;
}
break;
}
}

// Переходим к следующему триггеру
Trigger = Trigger->Next;
}
return 0; // Триггеры не найдены
}
```

### ***cTrigger::GetEnableState* и *cTrigger::Enable***

Функция **GetEnableState** проверяет текущее состояние триггера; вы передаете идентификационный номер триггера и получаете возвращенное состояние триггера. Если триггер отключен, вызов **GetEnableState** вернет **FALSE**. Если включен, вернет **TRUE**. Чтобы включить или выключить триггер, вызовите функцию **Enable**, используя в качестве первого аргумента идентификационный номер триггера.

Каждая из этих двух функций сканирует связанный список структур **sTrigger**. Функция **GetEnableState** возвращает значение флага активности первой найденной структуры из списка у которой идентификационный номер совпадает со значением, предоставленным в аргументе **ID**.

В функции **Enable** связанный список сканируется, и каждый экземпляр структуры, где идентификационный номер совпадает с переданным в аргументе **ID**, флаг активности устанавливается в значение, переданное в аргументе **Enable**. Посмотрите на код этих функций:

```
BOOL cTrigger::GetEnableState(long ID)
{
    sTrigger *TriggerPtr;
```

```

// Перебираем все триггеры, глядя на ID
if((TriggerPtr = m_TriggerParent) != NULL) {
    while(TriggerPtr != NULL) {

        // Если ID совпадает, возвращаем состояние
        if(TriggerPtr->ID == ID)
            return TriggerPtr->Enabled;

        // Переходим к следующему триггеру
        TriggerPtr = TriggerPtr->Next;
    }
}

// Возвращаем FALSE если ничего не нашли
return FALSE;
}

BOOL cTrigger::Enable(long ID, BOOL Enable)
{
    sTrigger *TriggerPtr;
    long Count = 0;

    // Перебираем все триггеры, глядя на ID
    if((TriggerPtr = m_TriggerParent) != NULL) {
        while(TriggerPtr != NULL) {

            // Если ID совпадает, устанавливаем флаг
            // и увеличиваем счетчик
            if(TriggerPtr->ID == ID) {
                TriggerPtr->Enabled = Enable;
                Count++;
            }

            // Переходим к следующему триггеру
            TriggerPtr = TriggerPtr->Next;
        }
    }

    // Возвращаем TRUE если какой-нибудь триггер изменен
    return (Count) ? TRUE : FALSE;
}

```

### ***cTrigger::GetNumTriggers* и *cTrigger::GetParentTrigger***

Как я люблю делать во всех моих классах, здесь я включаю две функции, которые вы можете использовать чтобы получить количество структур **sTrigger** в связанном списке, а также указатель на первую структуру (корневую, или родительскую, структуру) помещенную в список. Вы программируете эти две функции, **GetNumTriggers** и **GetParentTrigger**, следующим образом:

```

long cTrigger::GetNumTriggers()
{
    return m_NumTriggers;
}

sTrigger *cTrigger::GetParentTrigger()
{
    return m_TriggerParent;
}

```

## Использование триггеров

Как я обещал ранее, мы вновь возвращаемся к использованию файлов для хранения триггеров карты, на этот раз воспользовавшись классом **cTrigger**, созданным в разделе «Создание класса триггера». В этом разделе вы увидите как эффективно определять и загружать файл триггеров.

### ПРИМЕЧАНИЕ

Пример Mapping, находящийся на прилагаемом к книге CD-ROM (посмотрите в \BookCode\Chap13\Mapping) демонстрирует использование триггеров гораздо лучше, чем приведенный здесь небольшой пример. Не забудьте исследовать его!

### Определение файла триггеров

Начнем с примера файла данных триггеров (с именем test.trg):

```
1 0 1 -900 0 900 620
2 1 0 0 0 0 100 100 100
3 2 1 100 10 200 20 100
4 3 0 10 10 10 -100 -50 0 0 100
```

Первый триггер (ID# 1) — это сфера, расположенная в  $-900, 0, 900$  с радиусом 620. Второй триггер (ID# 2) — это куб, охватывающий область от 0, 0, 0 до 100, 100, 100. Третий триггер (ID# 3) является цилиндром, центр нижней грани которого находится в точке 100, 10, 200, с радиусом 20 и высотой 100 единиц. Четвертый триггер (ID# 4) — треугольник, охватывающий область от 10, 10 до 10,  $-100$  и до  $-50, 0$ ; координата Y (низ треугольной призмы) равна 0, и простирается он на 100 единиц вверх. Обратите внимание, что все остальные триггеры по умолчанию отключены.

### Загрузка файла триггеров

Чтобы загрузить файл триггеров, создайте экземпляр **cTrigger** и вызовите **Load**:

```
cTrigger Trigger;
Trigger.Load("test.trg");
```

### Касание триггера

И, наконец, чтобы увидеть, коснулся ли персонаж триггера, вызовите **GetTrigger** с координатами персонажа:

```
long TriggerID;

TriggerID = Trigger.GetTrigger(CharXPos, CharYPos, CharZPos);

if (TriggerID)
    MessageBox(NULL, "Trigger touched!", "Message", MB_OK);
```

Подпоясавшись этим чрезвычайно упрощенным примером загрузки и использования класса **cTrigger**, вы можете поработать с

демонстрационной программой Mapping, чтобы получить больше опыта в создании, загрузке и проверке столкновений персонажа с триггером с использованием класса **cTrigger**.

## Блокирование пути барьерами

В главе 8 я объяснил основы обнаружения столкновений. Вы знаете, как обнаружить, когда ходящий по карте персонаж сталкивается со стенами и обеспечить, чтобы он твердо стоял на земле. Но как быть с такими объектами, как двери, загораживающие путь вашему персонажу? Поскольку дверь не является частью ландшафта, я не включил двери, когда конструировал код обнаружения столкновений. Пришло время исправить эту ситуацию.

Объекты, мешающие свободному передвижению персонажа, называются *барьерами*. Барьеры могут находиться в двух состояниях: открытом (выключенном) или закрытом (включенном). Персонажам разрешено проходить через барьеры, когда они открыты, но не могут проходить через закрытые барьеры.

Вы можете рассматривать барьеры почти так же, как триггеры. Вы определяете барьер таким же способом, как определяете триггеры на карте. Можно определять сферические, кубические, цилиндрические и треугольные барьеры. У барьеров также есть состояние включения, где **TRUE** означает, что барьер блокирует перемещение персонажа, а **FALSE** означает, что проход через барьер свободен.

Большое отличие барьеров от триггеров в том, что у барьеров есть связанные с ними сетки и анимации. Это освобождает вас от бремени рисования барьеров и возлагает эту работу на движок барьеров. Все, что вам надо сделать, — назначить сетки и анимации.

Начнем использование барьеров с объявления класса барьера (он находится в файлах **Barrier.h** и **Barrier.cpp**, расположенных на CD-ROM в каталоге к главе 13), которое выглядит очень похожим на объявление класса триггера. Заметьте, что я также определяю перечисление и структуру (**sBarrier**), используемую для хранения данных каждого барьера:

```
enum BarrierTypes { Barrier_Sphere = 0, Barrier_Box,
                   Barrier_Cylinder, Barrier_Triangle };

typedef struct sBarrier {
    long Type;        // Сфера, куб и т.д.
    long ID;          // ID барьера
    BOOL Enabled;     // Флаг включения

    float XPos, YPos, ZPos; // Координаты
    float XRot, YRot, ZRot; // Вращение

    float x1, y1, z1; // Координаты 1
    float x2, y2, z2; // Координаты 2
    float x3, z3;     // Координаты 3
    float Radius;     // Радиус границы
```

Здесь сходство между триггерами и барьерами завершается. Барьеру необходимо графическое представление (трехмерная сетка), так что в последующем коде добавляется объект графического ядра **cObject**, который используется для хранения данных сетки и анимации барьера:

```
cObject Object; // Графический объект
```

Возвращаясь к сходству классов триггеров и барьеров, отметьте указатели для поддержки связанного списка, а также конструктор и деструктор структуры **sBarrier**:

```
sBarrier *Prev, *Next; // Связанный список

sBarrier() { Prev = Next = NULL; }
~sBarrier() { delete Next; Next = NULL; Object.Free(); }
} sBarrier;
```

Сходства между триггерами и барьерами продолжают проявляться и в объявлении класса барьера:

```
class cBarrier
{
private:
    cGraphics *m_Graphics; // Родительский объект cGraphics
    long      m_NumBarriers; // Кол-во барьеров в связанном списке
    sBarrier  *m_BarrierParent; // Связанный список барьеров

    long GetNextLong(FILE *fp); // Получение следующего long в файле
    float GetNextFloat(FILE *fp); // Получение следующего float в файле

    // Создание структуры sBarrier и вставка ее в список
    sBarrier *AddBarrier(long Type, long ID, BOOL Enabled,
                        float XPos, float YPos, float ZPos,
                        float XRot, float YRot, float ZRot);
};
```

Переключите ваше внимание на аргументы, которые получает функция **AddBarrier**. Помимо позиции, в которой располагается барьер (используются аргументы **XPos**, **YPos** и **ZPos**), функция **AddBarrier** получает значения вращения, используемые для рисования сетки барьера (аргументы **XRot**, **YRot** и **ZRot**, представляющие угол поворота в радианах по осям X, Y и Z, соответственно).

Обратите внимание на добавленные по всему классу барьера значения вращения, а также на добавление дополнительного трио координат, определяющего местоположение сетки в мире. Поскольку вы будете наталкиваться на эти дополнительные значения, я должен был упомянуть их.

Продолжим рассматривать объявление класса **cBarrier**:

```
public:
    cBarrier(); // Конструктор
    ~cBarrier(); // Деструктор

    // Функции для загрузки и записи списка барьеров
    BOOL Load(char *Filename);
    BOOL Save(char *Filename);
```

---

```
// Функции для установки сетки и анимации барьера
BOOL SetMesh(long ID, cGraphics *Graphics, cMesh *Mesh);
BOOL SetAnim(long ID, cAnimation *Anim,
             char *Name, long Time);
```

Когда вам надо назначить сетку барьеру, используйте функцию **SetMesh**, передав идентификационный номер барьера, а также объекты графического ядра **cGraphics** и **cMesh**. Для установки анимации барьера вы передаете идентификационный номер барьера, объект **cAnimation**, имя используемой анимации и время анимации (используя функции таймера, такие как **timeGetTime**).

После того, как вы назначили сетку и анимацию, можно визуализировать барьер на экране используя следующую функцию **Render**. Функция **Render** получает в качестве аргумента текущее время для обновления анимации (снова используем **timeGetTime**) и пирамиду видимого пространства, применяемую для отсечения тех барьеров, которые находятся вне нашего поля зрения:

```
// Визуализируем барьеры, используя
// заданную пирамиду видимого пространства
BOOL Render(unsigned long Time, cFrustum *Frustum);
```

Когда дело подходит к началу добавления барьеров в связанный список, класс **cBarrier** предлагает несколько предназначенных для этого функций, как это делал **cTrigger**. Взгляните на прототипы этих функций (я покажу вам, как работает каждая из них после того, как продемонстрирую объявление класса **cBarrier** целиком):

```
BOOL AddSphere(long ID, BOOL Enabled,
               float XPos, float YPos, float ZPos,
               float XRot, float YRot, float ZRot,
               float CXPos, float CYPos, float CZPos,
               float Radius);

BOOL AddBox(long ID, BOOL Enabled,
            float XPos, float YPos, float ZPos,
            float XRot, float YRot, float ZRot,
            float XMin, float YMin, float ZMin,
            float XMax, float YMax, float ZMax);

BOOL AddCylinder(long ID, BOOL Enabled,
                 float XPos, float YPos, float ZPos,
                 float XRot, float YRot, float ZRot,
                 float CXPos, float CYPos, float CZPos,
                 float Radius, float Height);

BOOL AddTriangle(long ID, BOOL Enabled,
                 float XPos, float YPos, float ZPos,
                 float XRot, float YRot, float ZRot,
                 float x1, float z1,
                 float x2, float z2,
                 float x3, float z3,
                 float CYPos, float Height);
```

```
    BOOL Remove(long ID);
    BOOL Free();

    long GetBarrier(float XPos, float YPos, float ZPos);

    BOOL GetEnableState(long ID);
    BOOL Enable(long ID, BOOL Enable);

    long GetNumBarriers();
    sBarrier *GetParentBarrier();
};
```

И снова, класс барьера очень похож на класс триггера, который вы видели в разделе «Создание класса триггера», так что я не буду зря тратить место, приводя здесь полный код класса **cBarrier**. Вместо этого обратитесь к классу **cTrigger** за спецификой большинства функций, и продолжайте чтение, чтобы увидеть устройство функций, являющихся особенностями класса барьера.

## **cBarrier::SetMesh и cBarrier::SetAnim**

В дополнение к графическому объекту, вам надо назначить сетку и анимацию. У каждого барьера есть выделенный **cObject**, используемый для ориентации, но сперва должна быть назначена сетка через функцию **SetMesh**. Далее следует установка анимации с помощью функции **SetAnim**. Взгляните на эти функции, отвечающие за установку сеток и анимаций:

```
BOOL cBarrier::SetMesh(long ID,
                       cGraphics *Graphics, cMesh *Mesh)
{
    sBarrier *BarrierPtr;
    long Count = 0;

    // Перебираем все барьеры, ища указанный ID
    if((BarrierPtr = m_BarrierParent) != NULL) {
        while(BarrierPtr != NULL) {

            // Если ID совпадает, устанавливаем сетку
            if(BarrierPtr->ID == ID) {
                BarrierPtr->Object.Create(Graphics, Mesh);
                Count++;
            }

            // Переход к следующему барьеру
            BarrierPtr = BarrierPtr->Next;
        }

        // Возвращаем TRUE, если установлена
        // какая-нибудь сетка
        return (Count) ? TRUE : FALSE;
    }

    BOOL cBarrier::SetAnim(long ID, cAnimation *Anim,
                           char *Name, long Time)
{
```

```

sBarrier *BarrierPtr;
long Count = 0;

// Перебираем все барьеры, ища указанный ID
if((BarrierPtr = m_BarrierParent) != NULL) {
    while(BarrierPtr != NULL) {

        // Если ID совпадает, устанавливаем анимацию
        if(BarrierPtr->ID == ID) {
            BarrierPtr->Object.SetAnimation(Anim, Name, Time);
            Count++;
        }

        // Переход к следующему барьеру
        BarrierPtr = BarrierPtr->Next;
    }
}

// Возвращаем TRUE, если установлена
// какая-нибудь анимация
return (Count) ? TRUE : FALSE;
}

```

После того, как вы загрузили или создали барьеры, назначьте сетки, используя идентификационные номера соответствующих барьеров. Обратите внимание, что функция **SetAnim** похожа на функцию **cObject::SetAnimation** — у вас есть имя анимации и стартовое время анимации.

Обе функции просто сканируют связанный список барьеров, ища соответствие идентификационного номера, после чего класс записывает сетку или устанавливает анимацию, и идет дальше по оставшейся части связанного списка.

## cBarrier::Render

Другой эксклюзивной функцией **cBarrier** (в противоположность **cTrigger**) является **Render**, получающая значение времени, используемое для обновления анимации барьера и пирамиду видимого пространства, применяемую для отбрасывания невидимых объектов барьеров. Взгляните на код функции **Render**:

```

BOOL cBarrier::Render(unsigned long Time, cFrustum *Frustum)
{
    sBarrier *BarrierPtr;
    float Radius;

    // Контроль ошибок
    if(Frustum == NULL)
        return FALSE;

    // Перебираем все барьеры
    if((BarrierPtr = m_BarrierParent) != NULL) {
        while(BarrierPtr != NULL) {

```



```
// Получаем радиус и проводим проверку
// пирамиды видимого пространства
BarrierPtr->Object.GetBounds(NULL, NULL, NULL, NULL,
                             NULL, NULL, &Radius);
if(Frustum->CheckSphere(BarrierPtr->XPos,
                        BarrierPtr->YPos,
                        BarrierPtr->ZPos, Radius)) {

    // Позиционирование объекта
    BarrierPtr->Object.Move(BarrierPtr->XPos,
                           BarrierPtr->YPos,
                           BarrierPtr->ZPos);
    BarrierPtr->Object.Rotate(BarrierPtr->XRot,
                              BarrierPtr->YRot,
                              BarrierPtr->ZRot);

    // Обновление анимации
    BarrierPtr->Object.UpdateAnimation(Time, TRUE);

    // Визуализация объекта
    BarrierPtr->Object.Render();
}

// Переходим к следующему барьеру
BarrierPtr = BarrierPtr->Next;
}

return TRUE;
}
```

В показанной выше функции **Render** сканируется связанный список барьеров и, для каждого барьера, выполняется проверка попадания в пирамиду видимого пространства. Если какая-либо часть барьера находится внутри пирамиды видимого пространства (даже если она скрыта другими объектами), обновляется соответствующая анимация и визуализируется сетка.

## Добавление барьеров с cBarrier

Класс барьера отмечает области на карте, используя геометрические фигуры, точно так же, как и класс триггера, но при этом класс барьера еще и позиционирует сетки. Взглянув снова на объявление класса **cBarrier**, обратите внимание, что каждая из функций добавления барьера — **AddSphere**, **AddBox**, **AddCylinder** и **AddTriangle** — имеет набор координат для позиционирования и вращения сетки барьера перед началом визуализации.

Чтобы определить, где будет расположена сетка, установите аргументы **XPos**, **YPos** и **ZPos** функции добавления барьера, указав где вы хотите визуализировать сетку. Вам также надо установить аргументы **XRot**, **YRot** и **ZRot**, указав значения поворота для рисования сетки.

Скажем, например, вы хотите добавить сферический барьер, которому уже назначена сетка. Барьер размещается по координатам 10, 20, 30 (с радиусом 40), а сетка помещается в 10, 0, 30 и не использует значения

вращения. Для добавления барьера вы вызываете **AddSphere** следующим образом:

```
cBarrier::AddSphere(1, TRUE,
                    10.0f, 0.0f, 30.0f, 0.0f, 0.0f, 0.0f,
                    10.0f, 20.0f, 30.0f, 40.0f);
```

Вы лучше разберетесь с добавлением и использованием барьеров в следующем разделе.

## Использование класса барьера

Использовать класс барьера не сложно; это очень похоже на использование класса триггера. Главное отличие в том, что вы добавляете к файлу данных барьера информацию о размещении объекта и назначаете соответствующие сетки и анимации.

### Создание файла данных барьеров

Файл данных барьеров организован также, как файл данных триггеров, но здесь определение каждого барьера содержит идентификационный номер, тип, флаг включения, координаты размещения ( $x$ ,  $y$ ,  $z$ ) и поворот (поворот по  $x$ , поворот по  $y$  и поворот по  $z$ ) для размещения графического объекта барьера. Заканчивается каждое определение данными, зависящими от типа барьера.

В следующем примере определяются два используемых барьера (находящихся в файле с именем **test.bar**). Координаты размещения и значения поворота барьера выделены полужирным шрифтом:

```
1 1 1 -900 0 0 0 0 0 -1154 0 10 -645 100 -10
2 1 0 0 0 -900 0 1.57 0 -10 0 -1154 10 100 -645
```

Здесь два барьера и оба имеют форму прямоугольного параллелепипеда. Графический объект первого барьера располагается в координатах  $-900, 0, 0$  и имеет значения вращения  $0, 0, 0$ . Первый параллелепипед охватывает область от  $-1154, 0, 10$  до  $-645, 100, -10$ .

Графический объект второго барьера располагается в координатах  $0, 0, -900$  и имеет значения вращения  $0, 1.57, 0$ . Второй барьер охватывает область от  $-10, 0, -1154$  до  $10, 100, -645$ .

### Загрузка данных барьеров

Чтобы загрузить и использовать файл данных барьеров, создайте экземпляр класса **cBarrier**, загрузите файл данных и соответствующие сетки и назначьте сетки:

```
// Graphics = ранее инициализированный объект cGraphics

cBarrier Barrier;

// Загрузка файла данных барьеров
Barrier.Load("test.bar");
```

```
// Загрузка используемых сеток и анимаций
cMesh Mesh;
cAnimation Anim;
Mesh.Load(&Graphics, "barrier.x");
Anim.Load("barrier.x", &Mesh);

// Назначение сеток и анимаций обеим загруженным барьерам
Barrier.SetMesh(1, &Graphics, &Mesh);
Barrier.SetMesh(2, &Graphics, &Mesh);
Barrier.SetAnim(1, &Anim, "AnimationName", 0);
Barrier.SetAnim(2, &Anim, "AnimationName", 0);
```

### ***Проверка столкновений с барьерами***

Чтобы увидеть, заблокирована ли область карты, вызовите **GetBarrier** с координатами персонажа. Если возвращается значение **TRUE**, проход заблокирован, и вы должны предпринять соответствующие действия. Возьмем следующий пример, который сравнивает координаты персонажа со всеми барьерами, загруженными из списка барьеров.

Вы используете трио значений, представляющих перемещение персонажа по каждой из осей, чтобы предварительно определить, заблокировано ли передвижение барьером. Скажем, персонаж перемещается на 10 единиц вдоль оси Z, и это значит, что в входящей переменной **ZMove** будет установлено значение 10. Эта переменная **ZMove** прибавляется к текущему местоположению персонажа, и, если произошло пересечение с барьером, переменная **ZMove** очищается, запрещая таким образом данное передвижение вдоль оси, как показано ниже:

```
// XPos, YPos, ZPos = координаты персонажа
// XMove, YMove, ZMove = значения перемещения персонажа
if(Barrier.GetBarrier(XPos+XMove, YPos+YMove, ZPos+ZMove) == TRUE) {
    // Проход запрещен, очищаем переменные перемещения
    XMove = YMove = ZMove = 0.0f;
}
```

### ***Визуализация барьеров***

Теперь вам надо только вызвать **cBarrier::Render**, чтобы нарисовать все объекты барьеров в поле зрения:

```
// Frustum = ранее инициализированный объект cFrustum
Barrier.Render(timeGetTime(), &Frustum);
```

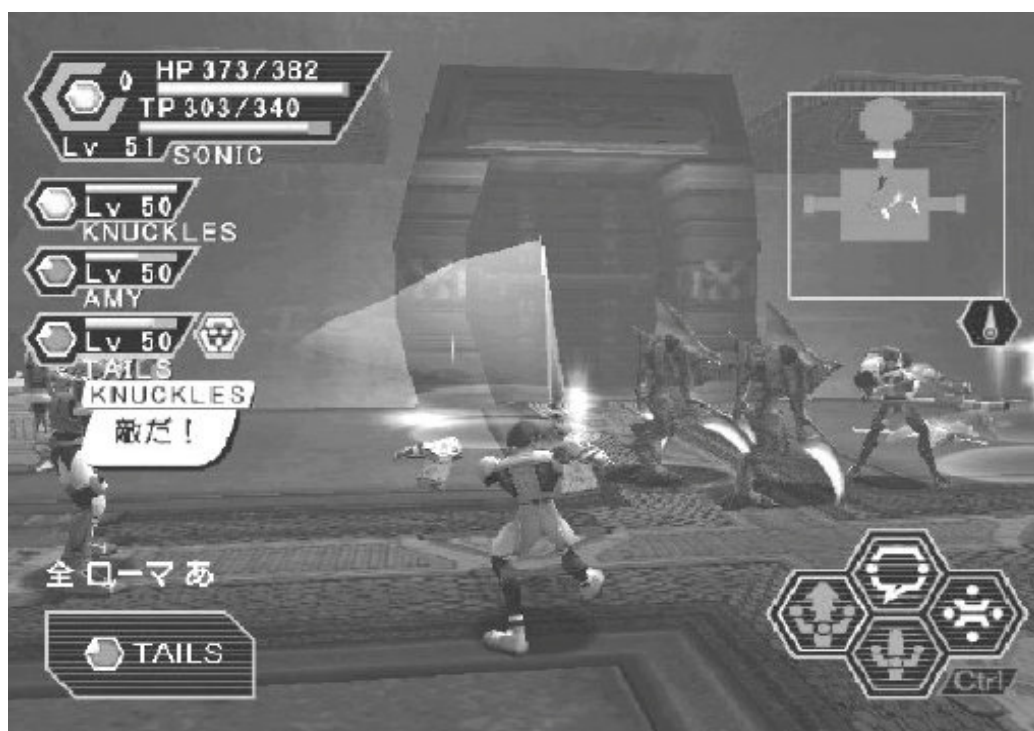
## **Использование автоматических карт**

Ваш игровой мир — огромное место, и когда игрок начнет исследовать свое окружение, вы можете захотеть облегчить ему жизнь, предоставив миниатюрную версию вашей карты в качестве путевого проводника. Подумайте не о простой карте, а о карте, достаточно умной, чтобы показывать, где игрок уже был и какие места еще нуждаются в исследовании.

Вам необходимо отображать только те части карты, которые игрок исследовал. Места, которые еще ни разу не были посещены, показывать не надо, то есть их следует показывать только тогда, когда они обнаружены игроками. Таким образом, игроки смогут оглянуться назад, где они уже были, и, возможно, взглянуть вперед и наметить пути для будущих экспедиций. Это волшебство называется автоматической картой.

## Автоматические карты в действии

Одна из моих любимых игр, Phantasy Star Online, выпущенная Sega, использует автоматические карты цельным образом. Посмотрите на рис. 13.5, который показывает рабочую автоматическую карту в верхнем правом углу экрана.



*Рис. 13.5. Phantasy Star Online использует плоскую двухмерную версию уровня, как показано выше*

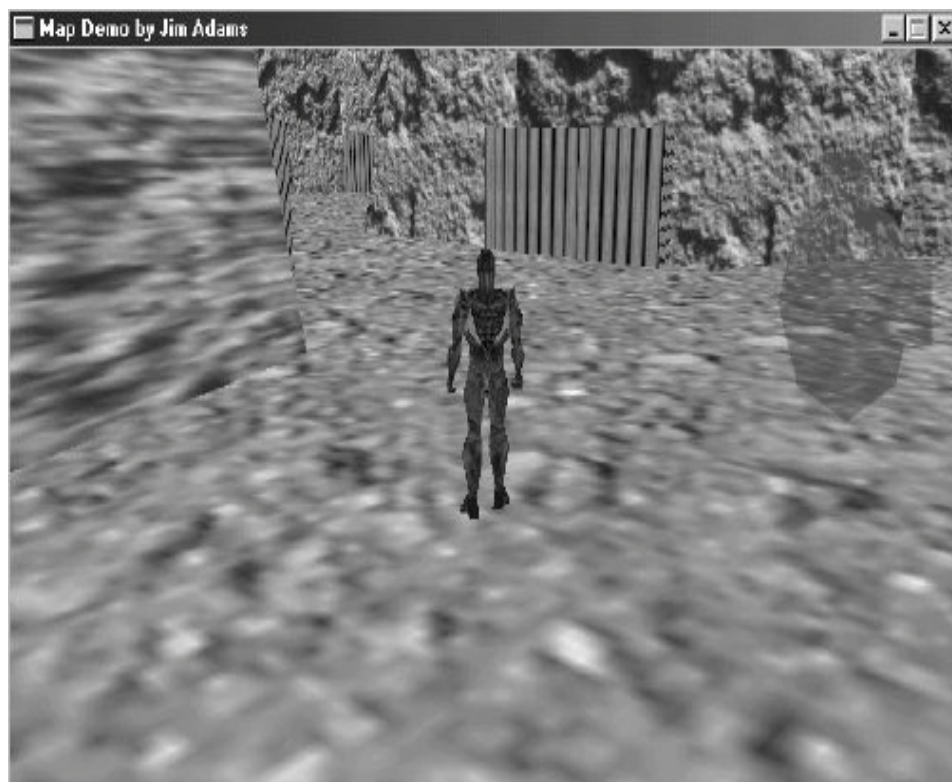
В Phantasy Star Online главный игрок и другие важные персонажи игры отображаются на автоматической карте как небольшие стрелки. Когда главный игрок ходит вокруг, карта прокручивается, чтобы показывать область вокруг игрока. Когда игрок посещает новую комнату (область), она появляется на автоматической карте.

Возможность автоматического картографирования, подобно многим другим возможностям, показанным в этой книге, легко воссоздать для ваших собственных игровых проектов.

## Большие карты, малые карты

Главный вызов здесь — преобразовать ваш большой игровой уровень в маленькую карту, подходящую для отображения в вашей игре. На рис. 13.6

показан снимок экрана демонстрационной программы Mapping. Обратите внимание на карту в верхнем правом углу экрана. Она использует альфа-смешивание (обратитесь к главе 2 за дополнительной информацией по этой теме), чтобы было видно игровое действие под ней.



*Рис. 13.6. Демонстрационная программа Mapping использует автоматическое картографирование, чтобы отображать области карты, которые игрок уже посетил*

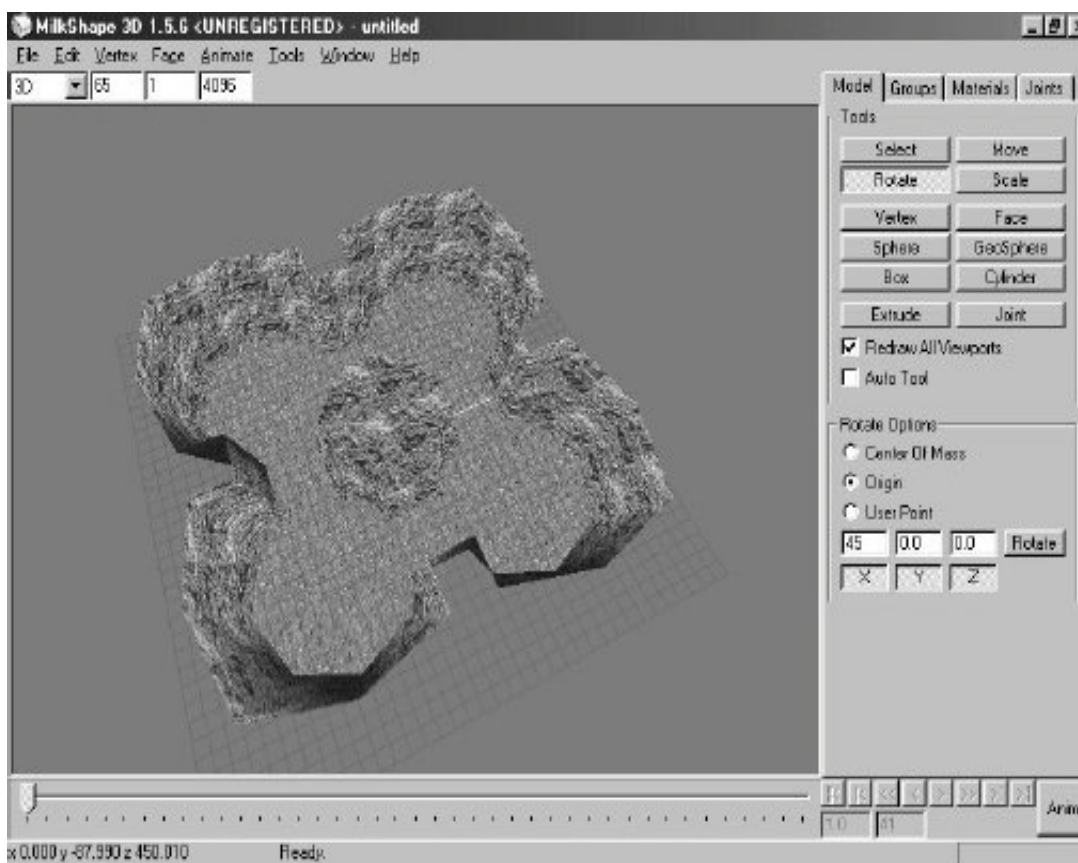
Простейший способ создать уменьшенную версию игрового уровня — перейти в редактор трехмерной графики и загрузить тот уровень, уменьшенную карту которого вы собираетесь сконструировать. На рис. 13.7 показан пример уровня, загруженный в MilkShape 3D и готовый для работы.

**ВНИМАНИЕ!**

MilkShape 3D неспособен показывать большие сетки, поэтому, чтобы рассмотреть сразу весь уровень, вам может потребоваться уменьшить масштаб сетки для работы, а затем вернуть прежний масштаб, когда вы будете записывать результат.

Для начала выберите все полигональные грани, нажав **Ctrl+A**. Щелкните по вкладке **Groups** и выберите **Regroup** для создания единой сетки. Затем щелкните по вкладке **Materials** и щелкайте **Delete**, пока не будут удалены все материалы. В результате вы должны получить единую сетку, которая не использует наложение текстур.

Теперь сложная часть: пройдите по всей сетке и удалите несущественные полигоны. В их число входят полигоны, которые никогда не будут видимы при взгляде сверху, которые используются для украшения, или представляют стены. Вам надо оставить только те полигоны, из которых состоит земля. Чтобы удалить грани, выберите ее (вам надо щелкнуть **Select**, а затем **Face**) и нажмите клавишу **Delete**.



*Рис. 13.7. Пример уровня, загруженный в редактор MilkShape 3D и готовый для преобразования в уменьшенную карту*

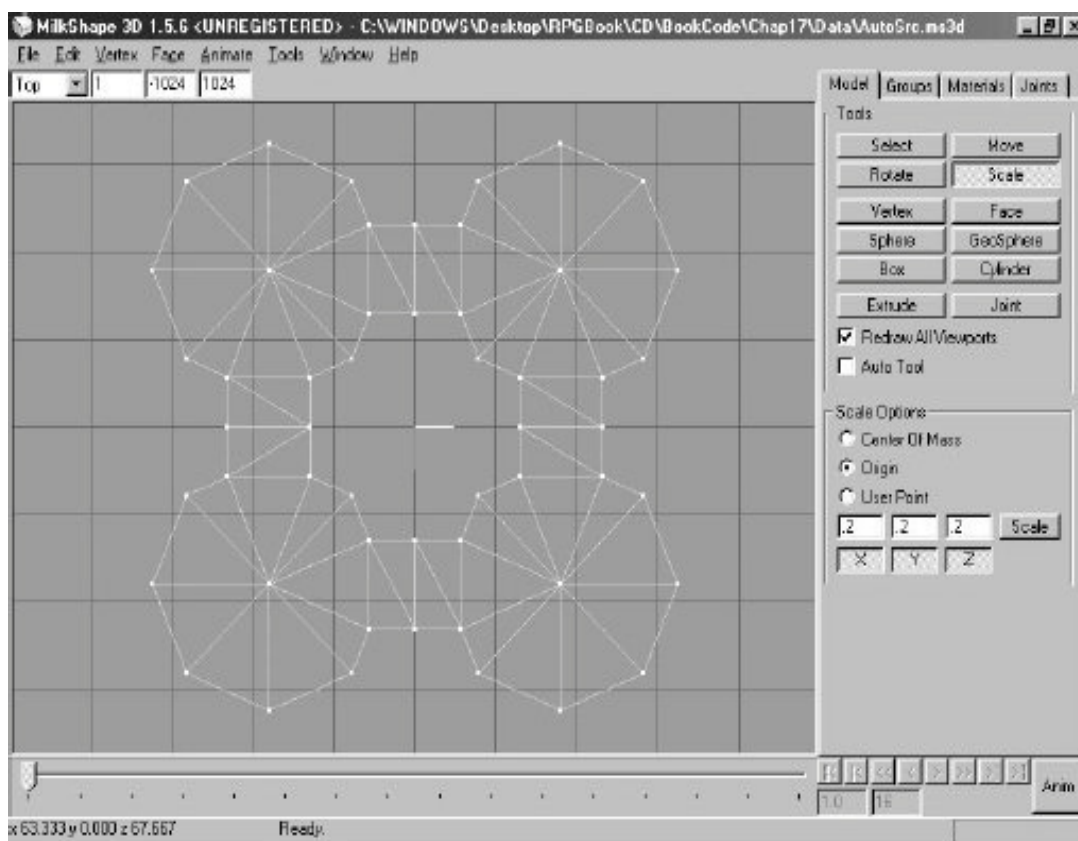
Если необходимо, создайте новые грани, чтобы сетка наилучшим образом соответствовала уровню, но избегайте перекрывающихся полигонов. Если вы используете альфа-смешивание для малой карты на дисплее, перекрывающиеся части полигонов будут более темными, что создаст плохой визуальный эффект (который вы можете видеть на рис. 13.8).



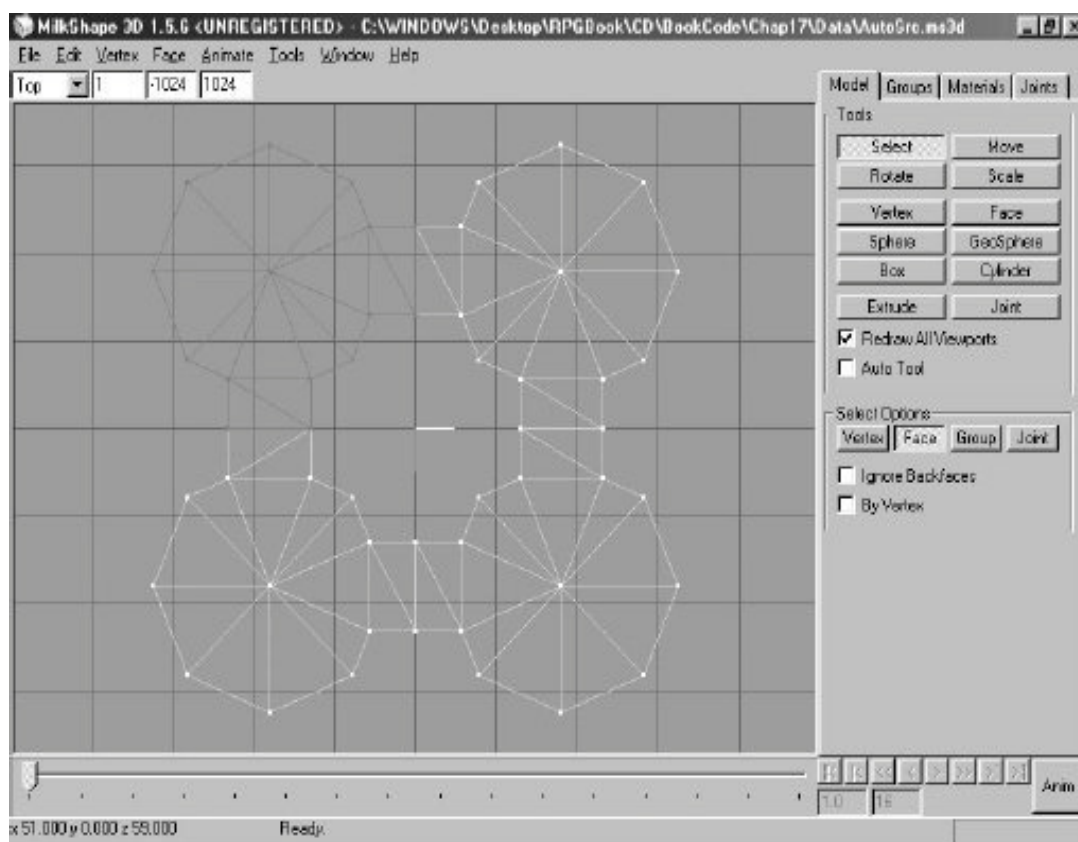
*Рис. 13.8. Перекрывающиеся полигоны на малой карте при визуализации с альфа-смешиванием могут создавать странные цветовые артефакты*

После небольшой переработки, как вы можете видеть на рис. 13.9, моя загруженная сетка уровня сократилась только до тех полигонов, которые доступны персонажам. Я добавил несколько новых полигонов, которые будут полезны при разделении сетки на несколько областей.





*Рис. 13.9. Малая карта содержит значительно меньше полигонов. Полигоны готовы к группировке в отдельные секции*



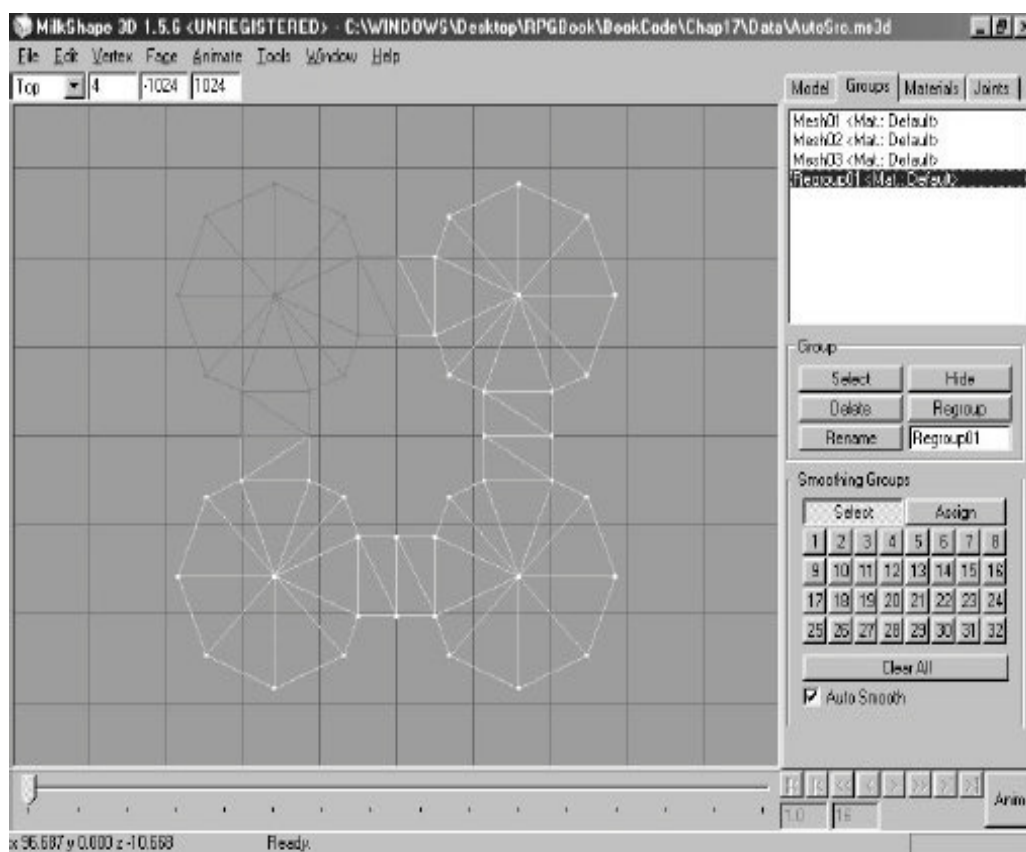
*Рис. 13.10. Выбор граней — первый этап построения отдельных областей карты*

И, наконец, карту надо разделить на меньшие фрагменты (которые, по сути, являются отдельными комнатами на карте), которые открываются по

мере того, как персонаж обнаруживает их. Начнем с отмены выделения (нажмите **Shift+Ctrl+A**). Теперь щелкните вкладку **Model**, щелкните **Select** и выберите **Face**.

Щелкайте по тем граням, которые вы хотите включить в один фрагмент карты. Как видно на рис. 13.10, я выбрал группу граней в верхнем левом углу сетки. Эти грани будут образовывать первую область карты.

Когда все нужные грани выбраны, снова щелкните по вкладке **Groups** и выберите **Regroup**. Обратите внимание, что создана новая группа (рис. 13.11).



*Рис. 13.11. Создана новая группа граней*

Поздравляю! Вы создали вашу первую область карты. Щелкните по группе **Regroup01**, снимите выбор со всех граней, и начинайте выбирать грани для следующего фрагмента карты. Продолжайте выбор граней и их перегруппировку, пока вся карта не будет разделена на различные группы.

Для карты примера я разбил сетку на четыре индивидуальных группы, представляющих верхний левый, верхний правый, нижний левый и нижний правый углы уровня. Итоговый файл карты вы найдете в примере программы из этой главы. Пойдемте дальше, запишем вашу карту и экспортируем ее в файл .X (используйте экспортер файлов .X, созданный мной и поставляемый на диске с этой книгой).



## Загрузка и отображение автоматической карты

Ну вот, уменьшенная карта создана и ждет использования. Сейчас вам надо загрузить файл .X и получить отдельные сетки, которые он содержит. Для загрузки сетки замечательно подходит использование объекта **cMesh** графического ядра.

Теперь вы конструируете массив буферов вершин — по одному буферу для каждой сетки автоматической карты. Вы заполняете каждый буфер вершин данными треугольных граней каждой сетки в объекте **cMesh**. Трюк здесь в том, что когда вы копируете данные вершин из сетки в буфер вершин, координата Y отбрасывается, чтобы получившийся буфер вершин сетки был плоским, обеспечивая двухмерный вид автоматической карты.

Для отображения загруженной карты вы просто позиционируете камеру, задаете порт просмотра для визуализации на дисплее и визуализируете каждый буфер вершин. При наличии автоматического картографирования вы пропускаете визуализацию буферов вершин, представляющих области карты еще не посещенные персонажем.

Хотя концепция кажется простой, перейдем к существу, взглянув на некоторый рабочий код.

## Создание класса автоматической карты

Класс автоматической карты, который я разработал для этой книги, загружает объект **cMesh** и сжимает его в плоскую версию карты. Плоская карта хранится в наборе буферов вершин. Эти буферы вершин используют только координаты X, Y и Z каждой вершины, плюс отдельный рассеиваемый цвет. Значит, карта будет компактной и простой для визуализации. Это также означает, что вы можете использовать альфа-смешивание, чтобы карта накладывалась на экран не скрывая происходящих важных игровых событий.

С каждой областью карты связан флаг, определяющий видима ли она. Класс позволяет вам включать и выключать этот флаг видимости, и гарантировать, что тяжелый труд игрока не пропадет напрасно, сохраняя и загружая эти флаги видимости. Хватит разговоров; посмотрите на объявление класса:

```
class cAutomap
{
    private:
        typedef struct {
            float x, y, z; // 3-D координаты
        } sGenericVertex;

        typedef struct {
            float x, y, z; // Координаты
            D3DCOLOR Diffuse; // Цвет карты
        } sVertex;

        #define AUTOMAPFVF (D3DFVF_XYZ | D3DFVF_DIFFUSE)
```

```

cGraphics *m_Graphics; // Родительский объект cGraphics

long m_NumSections; // Кол-во фрагментов карты
char *m_Visible;     // Видимость фрагментов

cVertexBuffer *m_MapVB; // Буфер вершин карты
cVertexBuffer m_PointerVB; // Буфер вершин указателя

D3DVIEWPORT9 m_Viewport; // Область для рисования карты
cCamera m_Camera; // Камера, используемая для рисования карты

float m_Scale; // Масштаб для рисования карты

public:
    cAutomap(); // Конструктор
    ~cAutomap(); // Деструктор

    // Функции для создания и освобождения карты
    BOOL Create(cGraphics *Graphics, char *Filename,
               long Color = D3DCOLOR_RGBA(64,64,64,255));
    BOOL Free();

    // Функции для сохранения/загрузки видимых областей карты
    BOOL Load(char *Filename);
    BOOL Save(char *Filename);

    // Возвращает количество фрагментов карты
    long GetNumSections();

    // Включение/выключение флага видимости фрагмента карты
    BOOL EnableSection(long Section, BOOL Enable);

    // Определение области для рисования карты
    BOOL SetWindow(long XPos, long YPos,
                  long Width, long Height);

    // Визуализация карты на экране
    BOOL Render(cCamera *OldCamera,
               float MXPos, float MYPos, float MZPos,
               float NumPositions,
               float *XPos, float *ZPos, float *Angle);
};

```

В начале вы видите определения двух структур данных вершин. Первую из них, **GenericVertex**, вы используете для доступа к координатам вершин исходной сетки. Вторая, **sVertex**, применяется для хранения фрагментов карты.

За структурами данных вершин следует набор переменных. Обратите внимание на объект **cGraphics**, используемый для загрузки сеток, количество используемых фрагментов карты, массив буферов вершин, массив переменных **char**, используемых для отметки видимых фрагментов карты, структуру данных порта просмотра, **cCamera**, переменную коэффициента масштабирования и буфер вершин указателя.

Вы должны свободно разобраться со всем этим, за исключением буфера вершин указателя и коэффициента масштабирования. Чтобы вам было

проще, масштаб загруженной карты уменьшается до пригодного для работы размера. Когда вы визуализируете автоматическую карту, вам необходимо задать координаты в масштабе большой карты, но класс автоматической карты уменьшает масштаб, чтобы он соответствовал уменьшенной карте.

Например, карта размером 1024 единицы в ширину и глубину масштабируется до 256 единиц в ширину и глубину. Фактически, карта масштабируется до размерв  $256 \times 256$ , независимо от ее размера в X-файле.

Что касается буфера вершин указателя, я добавил возможность отображать стрелки, представляющие каждый персонаж на карте. Стрелка указывает направление, в котором смотрит каждый персонаж. Буфер вершин просто содержит три точки и визуализируется красным цветом.

Помимо закрытых переменных класса вы также имеете дело с функциями.

### ***cAutomap::cAutomap* и *cAutomap::~~cAutomap***

Конструктор и деструктор класса **cAutomap** гарантируют, что все данные будут приведены в известное исходное состояние и что все используемые ресурсы будут освобождены. Конструктор только очищает некоторые переменные и ориентирует камеру, чтобы она была направлена вниз. Позднее вы используете эту камеру для визуализации карты. Деструктор вызывает функцию **Free** для освобождения всех используемых ресурсов. Взгляните на код этих функций:

```
cAutomap::cAutomap()
{
    m_Graphics = NULL;

    // Установка данных фрагментов и буфера вершин
    m_NumSections = 0;
    m_Visible = NULL;
    m_MapVB = NULL;

    // Направляем камеру вниз
    m_Camera.Rotate(1.57f, 0.0f, 0.0f);

    // Устанавливаем окно по умолчанию для отображения карты
    SetWindow(0,0,100,100);
    m_Scale = 1.0f; // Устанавливаем масштаб по умолчанию
}

cAutomap::~~cAutomap()
{
    Free();
}
```

### ***cAutomap::Create* и *cAutoMap::Free***

Глубоко вздохните, приступая к этой части. Функция **Create** самая большая из всех. Она загружает файл .X и преобразует каждую сетку из него в отдельный буфер вершин. Начнем с объявления ее переменных, исследуя переменные по одной, чтобы лучше понимать, что будет дальше:

```

BOOL cAutomap::Create(cGraphics *Graphics,
                     char *Filename, long Color)
{
    cMesh Mesh; // Загружаемый файл .X
    sMesh *MeshPtr; // Указатель на сетки в cMesh
    ID3DXMesh *IMeshPtr; // Указатель на сетку Direct3D
    sGenericVertex *GenVert; // Источник вершин
    sVertex Vertex, *VBPtr; // Местоназначение вершин
    long i, j, Num;
    long VertexSize, NumFaces;
    unsigned short *IndexPtr; // Указатель на буфер индексов сетки
    char *VertexPtr; // Указатель на буфер вершин сетки
    float Radius; // Радиус всех сеток в .X

    // Определение буфера вершин указателя
    sVertex PointerVerts = {
        { 0.0f, 0.0f, 10.0f, D3DCOLOR_RGBA(128,64,0,255) },
        { 5.0f, 0.0f, -10.0f, D3DCOLOR_RGBA(128,64,0,255) },
        { -5.0f, 0.0f, -10.0f, D3DCOLOR_RGBA(128,64,0,255) }
    };

    // Освобождаем предыдущую автоматическую карту
    Free();

    // Контроль ошибок
    if((m_Graphics = Graphics) == NULL || Filename == NULL)
        return FALSE;
}

```

К текущему моменту некоторые переменные объявлены, предыдущая автоматическая карта освобождена (через вызов **Free**), и проведен контроль ошибок. Обратите внимание, что в объявление переменных включено определение вершин для буфера вершин указателя.

Пойдем дальше, начав с кода, который загружает сетку карты, используемую для создания буферов вершин:

```

// Пытаемся загрузить сетку
if(Mesh.Load(Graphics, Filename) == FALSE)
    return FALSE;

// Получаем указатель на сетку
if((MeshPtr = Mesh.GetParentMesh()) == NULL) {
    Mesh.Free();
    return FALSE;
}

// Получаем размер вершин в исходной сетке
VertexSize = D3DXGetFVFVertexSize(MeshPtr->m_Mesh->GetFVF());

// Получаем ограничивающий радиус для масштабирования
Mesh.GetBounds(NULL, NULL, NULL, NULL, NULL, NULL, &Radius);
m_Scale = 128.0f / Radius;

// Получаем количество фрагментов в сетке карты
if(!(m_NumSections = Mesh.GetNumMeshes())) {
    Mesh.Free();
    return FALSE;
}

```

Первая задача состоит в загрузке файла **.X** с диска. Первая структура **sMesh** захватывается из объекта **cMesh** (вспомните из описания графического ядра, что класс **cMesh** хранит сетки в связанном списке структур **sMesh**).

Затем вы вычисляете размер структуры данных вершин, используемой в файле **.X**, и рассчитываете коэффициент масштабирования для изменения загруженной сетки. В конце вы сохраняете количество фрагментов карты в переменной класса. Заметьте, что количество фрагментов карты совпадает с числом сеток в файле **.X**.

Двигаясь дальше вы выделяете память для массива переменных **char**, каждый элемент которого сообщает, видим ли фрагмент карты. У каждого фрагмента карты есть соответствующий ему элемент массива. Вы также создаете массив буферов вершин (используя класс графического ядра **cVertexBuffer**). Эти буфера вершин будут использоваться для хранения фрагментов карты. Взгляните на код, который создает эти массивы и начинает сканирование списка сеток:

```
// Выделяем буфер видимости и очищаем его
m_Visible = new char[m_NumSections];
ZeroMemory(m_Visible, m_NumSections);

// Выделяем буферы вершин
m_MapVB = new cVertexBuffer[m_NumSections]();

// Перебираем каждую сетку в объекте cMesh
// и конструируем соответствующий буфер вершин.
// Убедитесь, что начинаете с последнего фрагмента карты,
// чтобы скомпенсировать для связанного списка
// упорядочение сеток в cMesh
Num = m_NumSections - 1;

while(MeshPtr != NULL) {
    // Получаем указатель на сетку
    IMeshPtr = MeshPtr->m_Mesh;
```

Вспомните, что сетки хранятся в связанном списке структур. Сейчас пришло время перебрать все структуры в связанном списке и запросить у каждой структуры указатель на реальный объект **Direct3D ID3DXMesh**, который содержит информацию сетки для отдельного фрагмента карты.

Затем вы блокируете буферы вершин и индексов (также как в главе 8) и начинаете извлекать данные вершин:

```
// Блокируем буферы индексов и вершин
IMeshPtr->LockIndexBuffer(D3DLOCK_READONLY,
                        (void**)&IndexPtr);
IMeshPtr->LockVertexBuffer(D3DLOCK_READONLY,
                        (void**)&VertexPtr);

// Создаем буфер вершин
NumFaces = IMeshPtr->GetNumFaces();
m_MapVB[Num].Create(Graphics, NumFaces*3,
                    AUTOMAPFVF, sizeof(sVertex));
```

```
// Блокируем буфер вершин
m_MapVB[Num].Lock(0, 0);
VBPtr = (sVertex*)m_MapVB[Num].GetPtr();
```

Создаваемый буфер вершин соответствует количеству полигональных граней в исходной сетке. Буфер вершин блокируется и будет получен указатель для начала сохранения вершин:

```
// Извлекаем вершины и конструируем список вершин
for(i = 0; i < NumFaces; i++) {
    for(j = 0; j < 3; j++) {
        // Получаем указатель на вершину
        GenVert = (sGenericVertex*)
            &VertexPtr[VertexSize * (*IndexPtr++)];

        // Создаем новые вершины
        Vertex.x = GenVert->x * m_Scale;
        Vertex.y = 0.0f;
        Vertex.z = GenVert->z * m_Scale;
        Vertex.Diffuse = Color;

        memcpy(VBPtr++, &Vertex, sizeof(sVertex));
    }
}
```

Два цикла перебирают каждую полигональную грань в исходной сетке и три вершины каждой грани копируются в буфер вершин карты. Обратите внимание, что вы используете только координаты X и Z, а координата Y устанавливается равной 0 (снова, чтобы сделать карту плоской). И, наконец, вы устанавливаете рассеиваемый цвет равным предоставленному цветовому значению (используется для визуализации карты).

```
// Разблокируем буфер вершин
m_MapVB[Num].Unlock();

// Разблокируем буферы
IMeshPtr->UnlockVertexBuffer();
IMeshPtr->UnlockIndexBuffer();

// Переходим к следующей сетке
Num--;
MeshPtr = MeshPtr->m_Next;
}
```

Вы завершаете процесс разблокируя буферы индексов и вершин исходной сетки и затем переходите к следующему фрагменту сетки карты в связанном списке сеток, загруженных из файла .X. Обратите внимание на переменную **Num**, отслеживающую создаваемые буферы вершин, уменьшаемую в показанном выше коде с каждой обработанной сеткой.

Вы уменьшаете переменную **Num**, а не увеличиваете, потому что сетки в объекте **cMesh** хранятся в обратном порядке (чтобы сделать загрузку быстрее), и вы должны компенсировать это, чтобы обеспечить последовательную нумерацию каждого фрагмента карты в соответствии с порядком хранения в файле .X.

Функция **Create** завершается созданием буфера вершин указателя и копированием в него определенных ранее данных описаний вершин. Исходная сетка освобождается и управление возвращается вызывавшему.

```
// Создание буфера вершин указателя персонажа
m_PointerVB.Create(Graphics, 3, AUTOMAPFVF, sizeof(sVertex));
m_PointerVB.Set(0, 3, &PointerVerts);

Mesh.Free(); // Освобождаем загруженную сетку

return TRUE;
}
```

Для удаления фрагментов карты из памяти вы предоставляете функцию **Free**, которая освобождает все выделенные ресурсы и подготавливает класс к загрузке другой карты:

```
BOOL cAutomap::Free()
{
    long i;

    // Освобождаем буферы вершин карты
    if(m_MapVB != NULL) {
        for(i = 0; i < m_NumSections; i++)
            m_MapVB[i].Free();

        delete [] m_MapVB;
        m_MapVB = NULL;
    }

    m_NumSections = 0; // Сбрасываем кол-во фрагментов
    delete [] m_Visible; // Освобождаем массив видимости
    m_Visible = NULL;
    m_PointerVB.Free(); // Освобождаем буфер вершин указателя

    return TRUE;
}
```

### ***cAutomap::Load и cAutomap::Save***

Вспомните, что вы должны включить каждый фрагмент карты, чтобы он был видим при визуализации. Массив **m\_Visible** отслеживает видимость каждого фрагмента карты; если значение элемента массива равно 0, соответствующий фрагмент карты не отображается. Если элемент установлен в 1, фрагмент карты рисуется.

В вашей игре, когда какие-нибудь фрагменты карты отмечаются как видимые, вы сохраняете эти флаги, чтобы игрок мог отслеживать свой прогресс в игре и позднее загрузить свою карту для продолжения игры. Вот функции загрузки и записи:

```
BOOL cAutomap::Load(char *Filename)
{
    FILE *fp;
    long Num;
    BOOL ReturnVal = FALSE;
```

```

// Открываем файл
if((fp = fopen(Filename, "rb")) == NULL)
    return FALSE;

// Получаем количество фрагментов
fread(&Num, 1, sizeof(long), fp);

// Проверяем соответствие и загружаем флаги видимости
if(m_NumSections == Num && m_Visible != NULL) {
    fread(m_Visible, 1, Num, fp);
    ReturnVal = TRUE;
}

fclose(fp);

return ReturnVal;
}

BOOL cAutomap::Save(char *Filename)
{
    FILE *fp;

    // Контроль ошибок
    if(m_NumSections && m_Visible == NULL)
        return FALSE;

    // Создаем файл
    if((fp = fopen(Filename, "wb")) == NULL)
        return FALSE;

    // Записываем количество фрагментов
    fwrite(&m_NumSections, 1, sizeof(long), fp);

    // Записываем флаги видимости
    fwrite(m_Visible, 1, m_NumSections, fp);

    fclose(fp); // Закрываем файл

    return TRUE; // Возвращаем флаг успеха
}

```

Формат хранения массива видимости простой: файл начинается с переменной **long**, указывающей количество элементов в массиве. Следом записывается весь массив видимости карты. Для загрузки массива видимости считывается количество элементов, проверяется, что оно соответствует загруженной в данный момент карте, после чего загружается массив.

### ***cAutomap::GetNumSections* и *cAutomap::EnableSection***

Эти две функции возвращают количество фрагментов загруженной карты и позволяют вам установить видимость каждого фрагмента карты. Фрагменты карты нумеруются последовательно в порядке их хранения в файле .X. Использование аргумента **TRUE** для **Enable** обеспечивает видимость фрагмента карты, а использование **FALSE** гарантирует, что фрагмент карты не будет отображаться. Держите это в уме, когда будете читать следующий код:



```
long cAutomap::GetNumSections()
{
    return m_NumSections;
}

BOOL cAutomap::EnableSection(long Section, BOOL Enable)
{
    if(Section >= m_NumSections || m_Visible == NULL)
        return FALSE;

    m_Visible[Section] = (Enable == TRUE) ? 1 : 0;

    return TRUE;
}
```

### ***cAutomap::SetWindow u cAutomap::Render***

Вы используете **SetWindow** чтобы определить область, в которой вы хотите отображать автоматическую карту (заданную в экранных координатах плюс высота и ширина в пикселах). Как видите, функция небольшая — она только инициализирует структуру порта просмотра, объявленную в классе **cAutomap**:

```
BOOL cAutomap::SetWindow(long XPos, long YPos,
                        long Width, long Height)
{
    m_Viewport.X = XPos;
    m_Viewport.Y = YPos;
    m_Viewport.Width = Width;
    m_Viewport.Height = Height;
    m_Viewport.MinZ = 0.0f;
    m_Viewport.MaxZ = 1.0f;

    return TRUE;
}
```

Что касается функции **Render**, здесь отображаются результаты ваших усилий. Для отображения карты вы предоставляете указатель на используемую в данный момент камеру (чтобы восстановить ее после смены матрицы вида), координаты камеры, используемой для визуализации карты, количество персонажей, которые будут показаны на карте, и три массива, определяющих координаты каждого персонажа и направление, в котором он смотрит, для рисования на автоматической карте:

```
BOOL cAutomap::Render(cCamera *OldCamera,
                    float MXPos, float MYPos, float MZPos,
                    float NumPositions,
                    float *XPos, float *ZPos, float *Angle)
{
    cWorldPosition Pos;
    D3DVIEWPORT9 OldViewport;
    long i;

    // Контроль ошибок
    if(m_Graphics == NULL || !m_NumSections ||
        m_MapVB == NULL || m_Visible == NULL)
```

```

return FALSE;

// Перемещение камеры
m_Camera.Move(MXPos*m_Scale, MYPos, MZPos*m_Scale);
m_Graphics->SetCamera(&m_Camera);

```

Функция **Render** начинается с определения нескольких переменных, контроля ошибок и установки камеры для визуализации фрагментов карты. Верно. Фрагменты карты остаются трехмерными сетками, только плоскими и видимыми сверху (по этой причине ранее в коде камера поворачивается вниз).

Затем вы создаете порт просмотра визуализации (параметры старого порта просмотра сохраняются для последующего восстановления). Вы устанавливаете режимы визуализации (нет Z-буферизации и нет текстур) и матрицу преобразования для центрирования автоматической карты в мире:

```

// Получаем старый порт просмотра и устанавливаем новый
m_Graphics->GetDeviceCOM()->GetViewport(&OldViewport);
m_Graphics->GetDeviceCOM()->SetViewport(&m_Viewport);

// Устанавливаем режимы визуализации и текстуру
m_Graphics->EnableZBuffer(FALSE);
m_Graphics->SetTexture(0, NULL);

// Визуализируем буферы вершин
m_Graphics->SetWorldPosition(&Pos);

```

Затем вы визуализируете каждый фрагмент карты. В действительности визуализируются только те фрагменты карты, которые помечены как видимые. Код для визуализации этих фрагментов карты небольшой, так что вы можете объединить его с кодом, который визуализирует указатели (представляющие местоположение персонажей на карте):

```

for(i = 0; i < m_NumSections; i++) {
    if(m_Visible[i])
        m_MapVB[i].Render(0, m_MapVB[i].GetNumVertices()/3,
                           D3DPT_TRIANGLELIST);
}

// Выключаем альфа-смешивание для визуализации указателей
m_Graphics->EnableAlphaBlending(FALSE);

// Рисуем местоположение персонажа
if(NumPositions) {
    for(i = 0; i < NumPositions; i++) {
        Pos.Move(XPos[i] * m_Scale, 0.0f, ZPos[i] * m_Scale);
        Pos.Rotate(0.0f, Angle[i], 0.0f);
        m_Graphics->gSetWorldPosition(&Pos);
        m_PointerVB.Render(0, 1, D3DPT_TRIANGLELIST);
    }
}

```

После визуализации фрагментов карты вы запрещаете альфа-смешивание (в том случае, если оно использовалось для визуализации карты) и размещаете и визуализируете буфер вершин указателя для каждого персонажа, переданного функции **Render**. Далее вы восстанавливаете

камеру и параметры порта просмотра, которые использовались перед визуализацией автоматической карты:

```
// Восстанавливаем старую камеру, если передана
if(OldCamera != NULL)
    m_Graphics->SetCamera(OldCamera);

// Восстанавливаем старый порт просмотра
m_Graphics->GetDeviceCOM()->SetViewport(&OldViewport);

return TRUE;
}
```

## Использование cAutomap

Демонстрационная программа к этой главе содержит замечательный пример использования класса автоматической карты, но для того, чтобы дать вам ясное представление о его использовании, вот небольшой пример. Начнем с создания экземпляра класса **cAutomap** и вызова функции **Create** для загрузки файла .X:

```
// Graphics = ранее инициализированный объект cGraphics
cAutomap Automap;

Automap.Create(&Graphics, "Map.x", D3DCOLOR_RGBA(64,64,64,255));
```

Теперь карта загружена и готова к использованию. Для визуализации карты используется темно-серый цвет (это причина появления макроса **D3DCOLOR\_RGBA**). Чтобы начать визуализацию карты вы должны сперва задать позицию окна, в котором будет выполняться визуализация:

```
Automap.SetWindow(0, 0, 200, 200); // Используем для карты окно
                                   // от 0,0 до 200,200
```

Затем вы отмечаете фрагмент карты как видимый:

```
Automap.EnableSection(0); // Делаем видимым 1-й фрагмент
```

И осталось только визуализировать карту:

```
Automap.Render(NULL, 0.0f, 200.0f, 0.0f, 0, NULL, NULL, NULL);
```

Показанный вызов помещает камеру в точку с координатами 0, 200, 0 (200 единиц над картой) и визуализирует единственный видимый фрагмент карты. А что насчет других фрагментов карты? Как ваша игра будет узнавать, какие фрагменты карты отметить как видимые? Используя триггеры, вот как!

Создав экземпляр класса **cTrigger**, вы можете установить на вашей карте триггеры, которые будут сигнализировать, что персонаж вошел в данную область карты и, следовательно, она должна быть помечена как видимая. Вы просто маркируете эти триггеры карты, используя те же самые идентификационные номера, что и для сеток фрагментов карты, хранящихся в файле .X карты (первой сетке в файле нужен триггер с идентификационным

номером 1, второй сетке нужен триггер с идентификационным номером 2 и т.д.).

Пример Mapping использует триггеры для отметки фрагментов карты, чтобы показать как персонажи входят в них — исследуйте пример, чтобы увидеть, о чем я говорю.

## Заканчиваем с картами и уровнями

В этой главе вы узнали, как использовать внешние файлы для хранения местоположения персонажей на вашей карте, как использовать триггеры и барьеры и как реализовать автоматическое составление карты. Это много информации для усвоения и я записал в вашу голову больше кода, чем вы, вероятно, сможете усвоить за один раз.

Убедитесь, что тщательно просмотрели код и точно представляете, как он работает, особенно, что относится к автоматическим картам. Основа кода проста, но попытка охватить сразу весь код класса может ошеломить. Убедитесь, что вы свободно владеете графическим ядром; показанный в этой главе код полагается на различные компоненты графического ядра, чтобы сделать ваши задачи программирования проще.

Если чувствуете себя достаточно смелым, попытайтесь создать класс персонажа, который обрабатывает загрузку персонажей, так же как это делает класс управления имуществом карты из главы 11, «Определение и использование объектов».

### Программы на CD-ROM

В каталоге \BookCode\Chap13 вы найдете следующие программы, демонстрирующие то, о чем вы читали в этой главе:

**Mapping** — программа демонстрирует автоматические карты, триггеры и барьеры. Перемещайте вашего персонажа вокруг, используя мышь и клавиши управления курсором. Используйте пробел чтобы открывать и закрывать двери. Местоположение: \BookCode\Chap13\Mapping\.



# Глава 14

## Создание боевых последовательностей

Размахивая мечом и швыряясь магией вы пробираетесь сквозь накатывающиеся волны демонических созданий. С каждой победой вы чувствуете, что становитесь более сильным; изучены новые заклинания и более мощное оружие готово к использованию. Враг начинает терять уверенность, и вот вы стоите на вершине горы, которую сами создали.

Так все и происходит, пока вы не сталкиваетесь с Красным Драконом, который до основания разрушил деревню. Весь ваш тяжелый труд оказывается тщетным за те 10 секунд, которые нужны огромному зверю, чтобы превратить вас в маленькие обугленные кусочки плоти.

Ах, но какое удовольствие — мощь и оживление, которое вы чувствуете уничтожая бесчисленных злодеев, и агония поражения, пришедшего от рук, или, в нашем случае, от когтей, более сильного противника. Удивительные боевые последовательности делают некоторые игры тем, чем они являются (или, по крайней мере, последовательности увеличивают удовольствие от игры). Теперь у вас есть шанс добавить боевые последовательности в ваши игры.

В этой главе вы узнаете как делается следующее:

- Проектирование внешних боевых последовательностей.
- Разработка простых боевых последовательностей.

## Проектирование внешних боевых последовательностей

Некоторые из моих самых любимых воспоминаний о ролевых играх связаны с битвами. Заметьте, не просто с любыми типами сражений. Я говорю о тех невероятных боевых последовательностях, которые сталкивают вас лицом к лицу со злыми ордами в стратегической войне. Если вы играли в Final Fantasy 7, то знаете, о каком виде боевых последовательностей я говорю.

Такие игры, как Final Fantasy 7, рассматривают сражения совсем под другим углом. Вместо того, чтобы сражаться в реальном времени, как это

делается в примере Chars (находящемся на CD-ROM в папке \BookCode\Chap12\Chars) из главы 12, «Управление игроками и персонажами», Final Fantasy 7 переключается на внешний экран боевой последовательности, где представление намного ближе к действию. Рис. 14.1 показывает типичную внешнюю боевую последовательность игры (из демонстрационной программы Battle, разработанной для этой главы; загляните в папку \BookCode\Chap14\Battle на CD-ROM).



*Рис. 14.1. Камера глядит на действие с фиксированной позиции, охватывая всех вовлеченных персонажей. На этом рисунке монстр мощным заклинанием атакует игрока*

Внешние боевые последовательности обычно следуют похожим проектам и шаблонам. Во-первых, у вас есть арена, являющаяся областью, представляющей небольшой фрагмент уровня, где сражаются персонажи. Персонажи располагаются на арене — игроки справа, враги слева.

#### **ПРИМЕЧАНИЕ**

Внешние боевые последовательности означают, что боевые последовательности отделены от последовательностей навигации в игре. Например, игрок, идущий по уровню, вступает в бой, и игра, в свою очередь, переключается на экран боевых последовательностей.

У каждого персонажа есть назначенный ему таймер (таймер зарядки), отслеживающий, как часто персонаж может выполнять действия. Таймер медленно увеличивается, пока не достигнет максимума, в этот момент персонаж может решить, какое действие выполнять. Больше никаких дерганий рукой и разбиваний кнопок — только размышление и осознанный выбор.

Как только действие и его жертва выбраны, персонажи медленно получают результаты. При произнесении заклинания персонаж выполняет длинные ритуалы, ясно показывающие удивительные заклинания и графические эффекты. Когда один персонаж атакует другого, двое быстро вступают в бой, независимо от расстояния между ними.

После того, как персонаж завершит действие, таймер зарядки персонажа сбрасывается, и персонаж должен снова ждать, чтобы быть полностью заряженным перед выполнением другого действия. Битва продолжается как было описано, пока все игроки или все враги не погибнут (или пока персонаж игрока не сбежит).

---

<b>ПРИМЕЧАНИЕ</b>	Большинство действий, такие как использование предметов, экипировка новым вооружением и броней, атака и произнесение заклинаний, требуют наличия полностью заряженного таймера игрока для выполнения действия.
-------------------	--

---

Когда сражение завершено, движок боевых последовательностей делит добычу и возобновляется обычный ход игры (естественно, до следующей боевой последовательности). Конечно, такой стиль сражений более медленный, чем битвы в реальном времени, но в целом это стоит ожидания.

## Техническая сторона

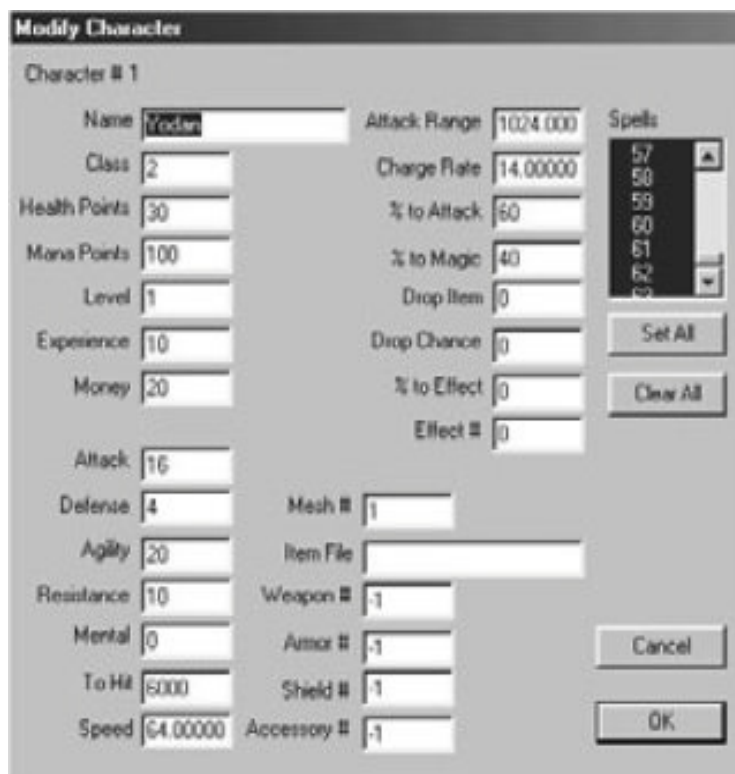
Арена представлена единой сеткой, также как персонажи и заклинания. Фактически, контроллеры персонажей и заклинаний, разработанные в главе 12, прекрасно подходят для боевых последовательностей. Вам надо только подстроить несколько вещей, чтобы эти контроллеры работали на вас.

Переработки требуют только дальность действия заклинаний и атак. Персонаж может атаковать любой другой персонаж, независимо от расстояния между ними. Аналогичным образом, заклинания можно нацеливать на любой персонаж, независимо от расстояния между ним и произносящим заклинание.

Чтобы обрабатывать различия в расстояниях, вы меняете данные главного списка персонажей и главного списка заклинаний. Персонажам нужна очень большая дальность атаки, поэтому вы запускаете Master Character List Editor и увеличиваете дистанцию атаки для каждого персонажа. На рис. 14.2 показан один персонаж, для которого параметр **Attack Range** установлен равным 1024.

Что касается заклинаний, вы увеличиваете максимальную дистанцию каждого из них в программе Master Spell List Editor. На рис. 14.3 показано одно из заклинаний, для которого я увеличил значение максимальной дистанции до 1024.





**Рис. 14.2.** Параметр каждого персонажа *Attack Range* должен быть увеличен в диалоговом окне *Modify Character* программы *MCL Editor* таким образом, чтобы каждый участвующий в сражении персонаж мог атаковать другого, независимо от расстояния между ними



**Рис. 14.3.** Диалоговое окно *Modify Spell* программы *MSL Editor* позволяет вам настроить дальность действия заклинаний

Помимо установки соответствующих расстояний, вы должны изменить параметры AI каждого персонажа при использовании контроллера персонажей. В боевых последовательностях вы должны заставлять персонажи стоять на месте, используя вариант AI **CHAR\_STAND**; иначе персонажи будут бродить по боевому уровню и могут даже покинуть его!

Если вы хотите позволить персонажам ходить по уровню, лучше всего использовать вариант AI для ходьбы по заданному маршруту (**CHAR\_ROUTE**), чтобы гарантировать, что персонаж будет перемещаться именно так, как вам нужно. Обратитесь к главе 12 за дополнительной информацией об использовании параметров AI и передвижении персонажей.

## Разработка боевых последовательностей

Что может быть лучше для изучения составления боевых последовательностей, чем пример для подражания? В этом разделе я покажу вам, как использовать информацию из главы 12 для создания своего собственного движка боевых последовательностей.

Чтобы увидеть, как я разрабатывал движок боевых последовательностей, скопируйте демонстрационный проект Battle с CD-ROM и изучайте его вместе с моими описаниями каждой функции из файла WinMain.cpp. Эти функции сравнительно небольшие, поскольку большую часть работы выполняют контроллер персонажей и контроллер заклинаний. Ваш боевой движок добавляет персонажи к драке, используя функцию **Add** контроллера персонажей, собирает и обрабатывает действия игрока и соответствующим образом обновляет персонажи.

Весь проект содержится внутри объекта класса **cApplication** из системного ядра. Вот объявление класса **cApplication** (наследуемого как **cApp**):

```
class cApp : public cApplication
{
    friend class cChars;

private:
    cGraphics      m_Graphics;          // Объект cGraphics
    cCamera        m_Camera;            // Объект cCamera
    cFont          m_Font;              // Объект cFont
    cWindow        m_Stats;             // Окно для состояния HP/MP
    cWindow        m_Options;           // Окно для заклинаний
    cInput         m_Input;             // Объект cInput
    cInputDevice   m_Keyboard;          // Объект ввода с клавиатуры
    cInputDevice   m_Mouse;            // Объект ввода от мыши
    cMesh          m_TerrainMesh;       // Сетка ландшафта
    cObject        m_TerrainObject;     // Объект ландшафта
    cVertexBuffer m_Target;            // Объект цели
    cTexture       m_Buttons;          // Кнопки и прочие рисунки

    // Контроллеры персонажей и заклинаний
    cCharacterController m_CharController;
    cSpellController m_SpellController;
```

```

sItem m_MIL[1024]; // Главный список предметов

// Смотрим, на какой персонаж указывает мышь
long GetCharacterAt(long XPos, long YPos);

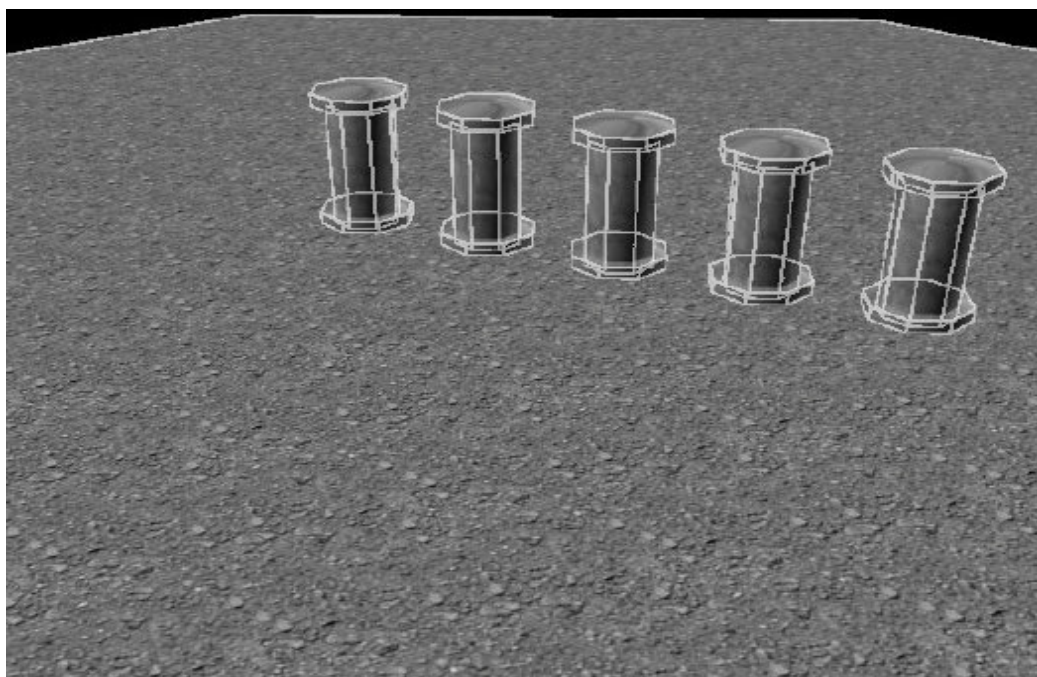
public:
    cApp();
    BOOL Init();
    BOOL Shutdown();
    BOOL Frame();
};

```

Класс приложения (**cApp**) выглядит довольно маленьким для такого амбициозного проекта, как боевые последовательности, но не позволяйте его размеру одурачить вас. Это мощный проект! Помните, что ранее разработанные контроллеры персонажей и заклинаний делают большую часть работы. Контроллеры персонажей и заклинаний идентичны тем, что вы видели в главе 12; даже код инициализации объектов тот же самый.

Единственное различие между этими персонажами и заклинаниями, и теми, что были в предыдущих примерах из этой книги, — дальность атаки и действия заклинаний. Все диапазоны действия атак и заклинаний были увеличены в главных списках до 1024, так что теперь не надо беспокоиться об их модификации в классе приложения.

В проекте Battle я сконструировал простую, небольшую сетку для арены (рис. 14.4) и загрузил ее в объекты **m\_TerrainMesh** и **m\_TerrainObject**. Вам не требуется дерево узлов, поскольку обычно сетка уровня точно подходит под размеры экрана. Сетки уровней боевых последовательностей не должны быть большими, так что проектируя свои собственные уровни, сохраняйте сетки достаточно небольшими, чтобы они помещались на экране.



*Рис. 14.4. Сетка арены для проекта Battle*

Управление прямолинейно. Чтобы ваш персонаж атаковал или произнес заклинание, вы должны сперва выбрать цель, на которую будет направлена атака или заклинание. Выбор цели осуществляется щелчком левой кнопки мыши по любому участвующему в битве персонажу. На экране возле выбранного персонажа появятся два маленьких вращающихся треугольника, отмечающие цель (они хранятся в буфере вершин **m\_Target**). Игрок может в любое время выбрать любой персонаж.

Когда у игрока полный заряд (заряд отображается медленно растущей полоской в нижнем правом углу экрана), он может выбрать тип выполняемого действия. Щелчок **Attack** заставит игрока ударить выбранную цель (наноса ей таким образом повреждения). Щелчок **Spell** открывает список известных заклинаний; щелчок по заклинанию приводит к его произнесению для выбранной цели. Помимо добавленного механизма выбора цели здесь нет ничего действительно нового. Итак, почему бы теперь не просмотреть код приложения?

## Глобальные данные

Класс приложения использует три глобальные переменные для хранения данных о сетках персонажей и заклинаний и информации об анимации персонажей:

```
#include "Core_Global.h"
#include "Window.h"
#include "Chars.h"
#include "WinMain.h"

// Глобальные имена сеток персонажей
char *g_CharMeshNames[] = {
    { "..\\Data\\Warrior.x" }, // Сетка # 0
    { "..\\Data\\Yodan.x" }   // Сетка # 1
};

sCharAnimationInfo g_CharAnimations[] = {
    { "Idle", TRUE },
    { "Walk", TRUE },
    { "Swing", FALSE },
    { "Spell", FALSE },
    { "Swing", FALSE },
    { "Hurt", FALSE },
    { "Die", FALSE },
    { "Idle", TRUE }
};

char *g_SpellMeshNames[] = {
    { "..\\Data\\Fireball.x" },
    { "..\\Data\\Explosion.x" },
    { "..\\Data\\Groundball.x" },
    { "..\\Data\\ice.x" },
    { "..\\Data\\bomb.x" },
};
```

Здесь используются те же самые персонажи и заклинания, что и в главе 12. Обратитесь к этой главе за сведениями об установке новых сеток персонажей и заклинаний.

## cApp::cApp

Вы используете конструктор класса приложения только для инициализации данных класса, что включает конфигурирование окна приложения:

```
cApp::cApp()
{
    m_Width   = 640;
    m_Height  = 480;
    m_Style   = WS_BORDER|WS_CAPTION|WS_MINIMIZEBOX|WS_SYSMENU;
    strcpy(m_Class, "BattleClass");
    strcpy(m_Caption, "Battle Demo by Jim Adams");
}
```

## cApp::Init

**Init**, первая перегруженная функция в классе приложения, инициализирует графическую систему и ввод, а также загружает всю графику, шрифты, предметы и другие данные, необходимые для программы:

```
BOOL cApp::Init()
{
    long i;
    FILE *fp;

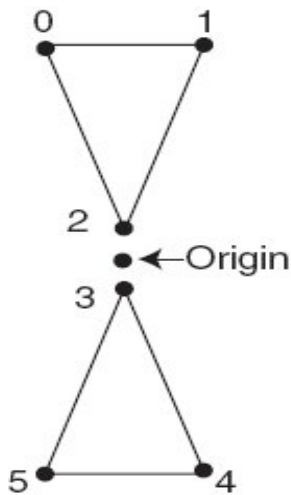
    // Инициализация графического устройства
    // и установка разрешения экрана
    m_Graphics.Init();
    m_Graphics.SetMode(GethWnd(), TRUE, TRUE);
    m_Graphics.SetPerspective(D3DX_PI/4, 1.3333f, 1.0f, 10000.0f);
    ShowMouse(TRUE);

    // Создание шрифта
    m_Font.Create(&m_Graphics, "Arial", 16, TRUE);
```

Как и в обычном графическом проекте, инициализируется графическая система и создается шрифт Arial. Затем идет инициализация системы и устройств ввода:

```
// Инициализация системы и устройств ввода
m_Input.Init(GethWnd(), GethInst());
m_Keyboard.Create(&m_Input, KEYBOARD);
m_Mouse.Create(&m_Input, MOUSE, TRUE);
```

Как я уже упоминал, вам нужен целевой персонаж для атаки или произнесения заклинания. Цель отмечает пара красных вращающихся треугольников, расположенных в содержащем шесть вершин буфере вершин со списком треугольников (как показано на рис. 14.5).



*Рис. 14.5. В соответствии с проектом буфер вершин указателя цели вращается относительно центральной координаты (точки отсчета)*

Сейчас функция **Init** создает буфер вершин, содержащий два треугольника, которые будут указывать на цель:

```
// Создаем буфер вершин указателя на цель
typedef struct {
    float x, y, z;
    D3DCOLOR Diffuse;
} sVertex;

sVertex Vert[6] = {
    { -20.0f,  40.0f, 0.0f, 0xFFFF4444 },
    {  20.0f,  40.0f, 0.0f, 0xFFFF4444 },
    {   0.0f,  20.0f, 0.0f, 0xFFFF4444 },
    {   0.0f, -20.0f, 0.0f, 0xFFFF4444 },
    {  20.0f, -40.0f, 0.0f, 0xFFFF4444 },
    { -20.0f, -40.0f, 0.0f, 0xFFFF4444 }
};

m_Target.Create(&m_Graphics, 6,
               D3DFVF_XYZ|D3DFVF_DIFFUSE,
               sizeof(sVertex));
m_Target.Set(0, 6, &Vert);
```

Когда буфер вершин указателя цели создан, необходимо с диска загрузить различную графику. Сперва вы загружаете изображения кнопок, используемых для выбора действия. Графические изображения этих кнопок представлены на рис. 14.6.



*Рис. 14.6. Изображения кнопок содержат кнопки **Attack** и **Spell**, а также полосу таймера зарядки*

Изображение, используемое для рисования таймера, зарядки объединено с графическими изображениями кнопок. Затем вы загружаете сетку и объект арены.

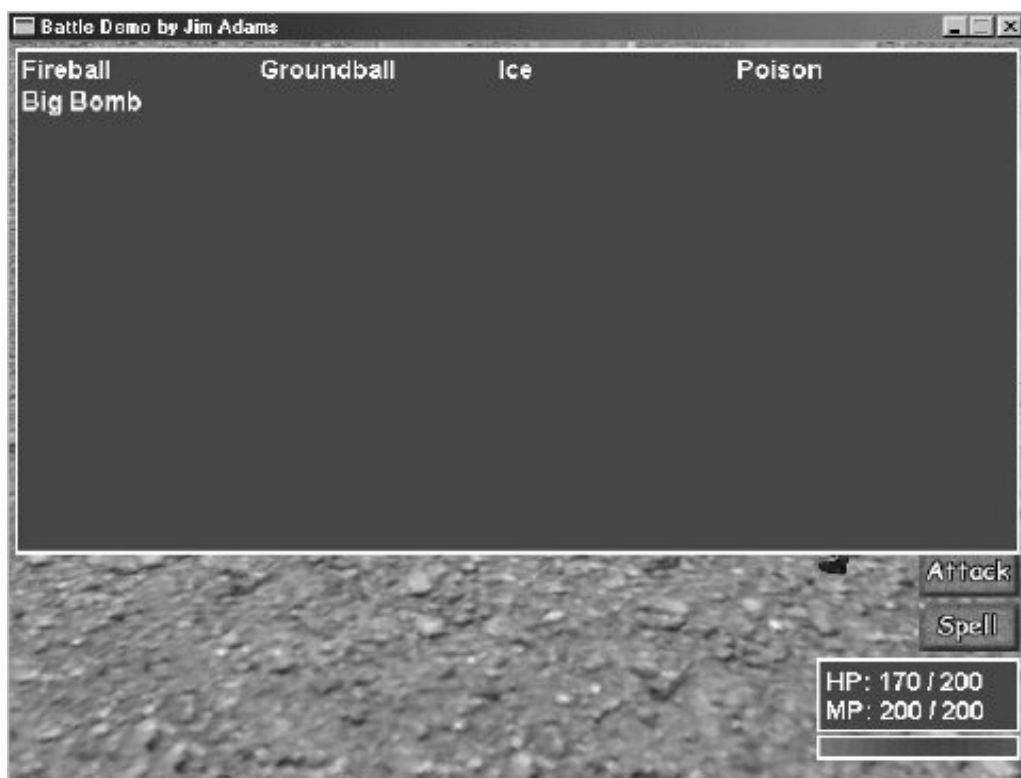
```
// Загрузка кнопок и другой графики
m_Buttons.Load(&m_Graphics, "..\\Data\\Buttons.bmp");

// Загрузка сетки ландшафта и установка объекта
m_TerrainMesh.Load(&m_Graphics, "..\\Data\\Battle.x",
                  "..\\Data\\");
m_TerrainObject.Create(&m_Graphics, &m_TerrainMesh);
```

Чтобы отображать состояние игрока (включающее очки здоровья и маны, как было определено в главе 12), вы создаете текстовое окно (**m\_Stats** — объект **cWindow** из главы 12) ниже персонажа. Второе окно (**m\_Options**) создается для отображения имен всех известных заклинаний, из которых игрок может выбирать необходимое заклинание. Как видно на рис. 14.7, это второе окно располагается поверх изображения.

```
// Создаем текстовые окна
m_Stats.Create(&m_Graphics, &m_Font);
m_Stats.Move(508, 400, 128, 48);

m_Options.Create(&m_Graphics, &m_Font);
m_Options.Move(4, 4, 632, 328);
```



*Рис. 14.7. Текстовые окна позволяют приложению показывать состояние игрока и все известные заклинания*

Затем вам надо загрузить главный список предметов и инициализировать классы контроллеров персонажей и заклинаний. Код инициализации контроллеров идентичен используемому в демонстрационном проекте **Chars** из главы 12.

```

// Загрузка главного списка предметов
for(i = 0; i < 1024; i++)
    ZeroMemory(&m_MIL[i], sizeof(sItem));

if((fp=fopen("../Data\\Default.mil", "rb")) != NULL) {
    for(i = 0; i < 1024; i++)
        fread(&m_MIL[i], 1, sizeof(sItem), fp);
    fclose(fp);
}

// Инициализация контроллера персонажей
m_CharController.Init(&m_Graphics, NULL, &m_Font,
    "../Data\\Default.mcl", (sItem*)&m_MIL,
    m_SpellController.GetSpell(0),
    sizeof(g_CharMeshNames) / sizeof(char*),
    g_CharMeshNames,
    "../Data\\", "../Data\\",
    sizeof(g_CharAnimations)/sizeof(sCharAnimationInfo),
    (sCharAnimationInfo*)&g_CharAnimations,
    &m_SpellController);

// Инициализация контроллера заклинаний
m_SpellController.Init(&m_Graphics, NULL,
    "../Data\\Default.msl",
    sizeof(g_SpellMeshNames) / sizeof(char*),
    g_SpellMeshNames,
    "../Data\\", &m_CharController);

```

И, завершая функцию **Init**, вы размещаете на арене несколько персонажей (игрока и монстров). (Здесь меня одолела лень; жесткое программирование участвующих персонажей и их местоположения следует заменить на случайный выбор, но я оставляю это для вас.) Чтобы добавить персонаж к сражению и разместить его, используйте функцию **Add** контроллера персонажей, как показано ниже:

```

// Добавим персонаж игрока
m_CharController.Add(0, 0, CHAR_PC, CHAR_STAND,
    200.0f, 0.0f, 0.0f, 4.71f);

// Все остальные персонажи жестко запрограммированы
m_CharController.Add(1, 1, CHAR_MONSTER, CHAR_STAND,
    -200.0f, 0.0f, 0.0f, 1.57f);
m_CharController.Add(2, 1, CHAR_MONSTER, CHAR_STAND,
    -100.0f, 0.0f, -200.0f, 1.57f);
m_CharController.Add(3, 1, CHAR_MONSTER, CHAR_STAND,
    0.0f, 0.0f, 100.0f, 1.57f);

// Дадим одному из монстров топор
m_CharController.Equip(m_CharController.GetCharacter(1),
    8, WEAPON, TRUE);

return TRUE;
}

```

Обратите внимание, что всего в драке участвуют четыре персонажа. Чтобы игроку было труднее, я пошел дальше и экипировал второго персонажа (монстра) топором. Чтобы экипировать персонажа оружием, используется функция контроллера персонажей **Equip**, как я показал.



## cApp::Shutdown

Функция **cApp::Shutdown** (являющаяся противоположностью **cApp::Init**) освобождает все используемые ресурсы (сетки, объекты и контроллеры) как показано ниже:

```
BOOL cApp::Shutdown()
{
    // Освобождаем контроллеры
    m_CharController.Free();
    m_SpellController.Free();

    // Освобождаем объекты и сетки
    m_TerrainMesh.Free();
    m_TerrainObject.Free();

    // Освобождаем окна
    m_Stats.Free();
    m_Options.Free();

    // Освобождаем буфер вершин указателя цели
    m_Target.Free();
    m_Buttons.Free();

    // Отключаем ввод
    m_Keyboard.Free();
    m_Mouse.Free();
    m_Input.Shutdown();

    // Отключаем графику
    m_Font.Free();
    m_Graphics.Shutdown();

    return TRUE;
}
```

## cApp::Frame

Функция **cApp::Frame** в ее текущем воплощении заметно разрослась. Здесь **Frame** выполняет работу по сбору и обработке пользовательского ввода. Для ввода используется только мышь; левая кнопка мыши выбирает целевой персонаж, произнесение заклинания или выполнение атаки. Правая кнопка мыши закрывает окно выбора заклинания, если оно открыто. Как только выбраны цель и действие, выполняется соответствующая обработка.

Помимо обработки ввода, функция **Frame** визуализирует все на экране, включая арену, персонажей, анимацию заклинаний, окно состояния (показывающее очки здоровья и маны игрока), полосу зарядки, список заклинаний и кнопки действий:

```
BOOL cApp::Frame()
{
    static DWORD UpdateCounter = timeGetTime();
    static sCharacter *PCChar=m_CharController.GetCharacter(0);
    static BOOL SelectSpell = FALSE;
    static long TargetID = -1;
```

```

cWorldPosition Pos;
sCharacter *CharPtr;
sSpell *SpellPtr;
char Text[128];
long x, y, Num, i;
float MinY, MaxY, YOff;

// Ограничиваем до 30 fps
if(timeGetTime() < UpdateCounter + 33)
    return TRUE;
UpdateCounter = timeGetTime();

```

Таймер кадров управляет скоростью, с которой идет игра. В рассматриваемом проекте она ограничена 30 кадрами в секунду, что означает, что обновление кадра выполняется каждые 33 миллисекунды. Если обновление разрешено, исполнение продолжается и переходит к чтению пользовательского ввода.

---

**ПРИМЕЧАНИЕ** Чтобы вычислить количество миллисекунд ожидания в вашей игре, разделите 1000 на используемое значение частоты кадров. В данном случае 1000 делим на 30 и получаем 33,333333, округляемое до 33.

---

```

// Чтение ввода
m_Keyboard.Acquire(TRUE);
m_Keyboard.Read();
m_Mouse.Acquire(TRUE);
m_Mouse.Read();

// Выход, если нажата ESC
if(m_Keyboard.GetKeyState(KEY_ESC) == TRUE)
    return FALSE;

```

Затем **Frame** определяет, что делать, если игрок нажал левую кнопку мыши. Помните, что вы можете выбирать цель, произносить заклинание или атаковать:

```

// Получаем выбранного персонажа, если нажата левая кнопка
if(m_Mouse.GetButtonState(MOUSE_LBUTTON) == TRUE) {
    // Получаем координаты указателя мыши
    x = m_Mouse.GetXPos();
    y = m_Mouse.GetYPos();

    // Блокируем кнопки мыши
    m_Mouse.SetLock(MOUSE_LBUTTON, TRUE);
    m_Mouse.SetButtonState(MOUSE_LBUTTON, FALSE);
}

```

Далее **Frame** определяет, что обрабатывать. Если игрок щелкнул по кнопке **Spell**, открывается окно списка заклинаний (**m\_Options**), показывающее все известные заклинания. Когда окно списка заклинаний открыто, флаг **SelectSpell** (который по умолчанию установлен в **FALSE**) устанавливается в **TRUE** и **Frame** ждет, пока будет выбрано заклинание или закрыто окно (по щелчку правой кнопкой мыши).

```

// Смотрим, выбираем ли заклинание
if(SelectSpell == TRUE) {
    // Получаем указатель на заклинание
    Num = ((y-8)/20) * 4 + ((x-8)/150);
}

```

Вы выбираете заклинание, вычисляя координаты щелчка мыши на экране и сравнивая их с местоположением, где в окне печатается каждое из названий заклинаний. Каждое название заклинания занимает область окна  $150 \times 20$  пикселей (сами области начинаются с экранных координат 8, 8), что дает место, достаточное для размещения в окне 64 названий заклинаний.

```
// Проверяем, знает ли игрок заклинание
// (и достаточно ли у него маны)
if(Num >= 0 && Num < 64) {
    SpellPtr = m_SpellController.GetSpell(Num);
    if(PCChar->Def.MagicSpells[Num/32] & (1<<(Num & 31)) &&
        SpellPtr->Name[0] &&
        PCChar->ManaPoints >= SpellPtr->Cost) {
        // Устанавливаем номер произносимого заклинания
        PCChar->SpellNum = Num;
        // Устанавливаем цель
        PCChar->SpellTarget = CHAR_MONSTER;
        m_CharController.SetAction(PCChar, CHAR_SPELL);
        SelectSpell = FALSE; // Закрываем окно выбора
    }
}
```

Как только игрок выбирает заклинание в окне со списком заклинаний, соответствующее заклинание начинает действовать и класс контроллера обрабатывает его эффекты. Окно выбора заклинаний помечается как закрытое и исполнение продолжается. Если выбор заклинания еще не был начат, функция **Frame** продолжает выполнение, проверяя, не щелкнул ли игрок по кнопкам **Attack** или **Spell**.

```
}
}
} else {
    // Смотрим, щелкнули ли по кнопке
    // (если выбрана цель и закончен заряд)
    if(TargetID != -1 && PCChar->Charge >= 100.0f) {
        // Устанавливаем сведения о жертве и атакующем
        CharPtr = m_CharController.GetCharacter(TargetID);
        PCChar->Victim = CharPtr;
        CharPtr->Attacker = PCChar;
        PCChar->TargetX = CharPtr->XPos;
        PCChar->TargetY = CharPtr->YPos;
        PCChar->TargetZ = CharPtr->ZPos;

        // Определяем, выбрана ли атака,
        // сравнивая координаты указателя мыши с
        // координатами кнопки Attack на экране,
        // находящимися в диапазоне 572,328 - 636, 360
        if(x >= 572 && x < 636 && y >= 328 && y < 360)
            m_CharController.SetAction(PCChar, CHAR_ATTACK);

        // Определяем, выбрано ли произнесение заклинания,
        // сравнивая координаты указателя мыши с
        // координатами кнопки Spell на экране,
        // находящимися в диапазоне 572,364 - 636, 396
        if(x >= 572 && x < 636 && y >= 364 && y < 396)
            SelectSpell = TRUE;
    }
}
```

После того, как действие выбрано (и если указана цель и таймер зарядки игрока достиг максимума), устанавливается информация об атаке. Если

игрок щелкает по кнопке **Attack**, инициируются действия атаки. Если игрок щелкает по кнопке **Spell**, открывается окно выбора заклинания.

Независимо от того, по какому элементу управления щелкнул игрок, выполнение кода продолжается, и следующие строки с помощью вызова **GetCharacterAt** определяют, был ли выбран какой-нибудь целевой персонаж:

```
// Смотрим, выбран ли персонаж
TargetID = GetCharacterAt(x, y);
}

// Очищаем состояние заклинания, если нажата правая кнопка мыши
if(m_Mouse.GetButtonState(MOUSE_RBUTTON) == TRUE) {
    // Блокируем кнопки мыши
    m_Mouse.SetLock(MOUSE_RBUTTON, TRUE);
    m_Mouse.SetButtonState(MOUSE_RBUTTON, FALSE);
    SelectSpell = FALSE;
}

// Обновляем контроллеры
m_CharController.Update(33);
m_SpellController.Update(33);
```

Если игрок нажал правую кнопку мыши, окно выбора заклинания закрывается и исполнение функции **Frame** продолжается обновлением контроллеров персонажей и заклинаний (путем вызова функции **Update** каждого из контроллеров). К этому моменту вся обработка ввода завершена и начинается визуализация.

```
// Установка камеры
m_Camera.Point(300.0f, 300.0f, -340.0f, 0.0f, 0.0f, 0.0f);
m_Graphics.SetCamera(&m_Camera);

// Визуализируем все
m_Graphics.Clear(D3DCOLOR_RGBA(0,32,64,255));
if(m_Graphics.BeginScene() == TRUE) {
```

Камера установлена, сцена очищена и начинается визуализация сцены путем рисования арены, персонажей и заклинаний.

```
// Визуализация ландшафта
m_Graphics.EnableZBuffer(TRUE);
m_TerrainObject.Render();

// Визуализация всех персонажей
m_CharController.Render();

// Визуализация заклинаний
m_SpellController.Render();

// Проверяем, надо ли визуализировать указатель цели
if(TargetID != -1) {
    // Перемещаем указатель цели в
    // место нахождения персонажа
    CharPtr = m_CharController.GetCharacter(TargetID);
    Pos.EnableBillboard(TRUE);
```

```
Pos.Move(CharPtr->XPos, CharPtr->YPos, CharPtr->ZPos);
Pos.Rotate(0.0f, 0.0f, (float)timeGetTime() / 100.0f);

// Смещаем на половину высоты персонажа
CharPtr->Object.GetBounds(NULL, &MinY, NULL,
                          NULL, &MaxY, NULL, NULL);
YOff = MinY + ((MaxY - MinY) * 0.5f);
Pos.MoveRel(0.0f, YOff, 0.0f);

// Визуализируем указатель цели
m_Graphics.SetTexture(0, NULL);
m_Graphics.EnableZBuffer(FALSE);
m_Graphics.SetWorldPosition(&Pos);
m_Target.Render(0, 2, D3DPT_TRIANGLELIST);
m_Graphics.EnableZBuffer(TRUE);
}
```

После того, как визуализированы арена, персонажи и заклинания, приходит время визуализировать буфер вершин указателя цели (только если цель выбрана). Вы центрируете указатель цели на середине персонажа (основываясь на его высоте) и рисуете указатель (используя матрицу мирового преобразования щита, чтобы указатель цели всегда был обращен к камере).

Затем приходит черед окна состояния игрока, в котором вы обновляете отображение текущих очков здоровья и маны игрока:

```
// Отображение экрана состояния
sprintf(Text, "HP: %ld / %ld\r\nMP: %ld / %ld",
        PCChar->HealthPoints, PCChar->Def.HealthPoints,
        PCChar->ManaPoints, PCChar->Def.ManaPoints);
m_Stats.Render(Text);
```

Затем вы отображаете таймер зарядки. Значение таймера зарядки находится в диапазоне от 0 до 100, и используется ранее загруженное в объект **m\_Buttons** изображение кнопок, из которого вырезается только небольшая часть изображения, представляющая текущий уровень заряда.

```
// Отображаем измеритель заряда
m_Graphics.BeginSprite();
m_Buttons.Blit(508, 450, 0, 64, 128, 16);
m_Buttons.Blit(510, 452, 0, 80,
              (long)(1.24f*PCChar->Charge), 12);
m_Graphics.EndSprite();
```

Теперь рисуем кнопки действий (только если таймер заряда достиг максимального значения):

```
// Отображение вариантов атаки
if(m_CharController.GetCharacter(0)->Charge >= 100.0f) {
    m_Graphics.BeginSprite();
    m_Buttons.Blit(572, 328, 0, 0, 64, 32);
    m_Buttons.Blit(572, 364, 0, 32, 64, 32);
    m_Graphics.EndSprite();
}
```

В завершение визуализации, если необходимо, рисуем окно выбора заклинаний, и отображаем каждое из известных заклинаний, чтобы игрок мог сделать свой выбор:

```
// Отображаем список заклинаний
if(SelectSpell == TRUE) {
    m_Options.Render();

    // Отображаем известные заклинания
    for(i = 0; i < 64; i++) {
        SpellPtr = m_SpellController.GetSpell(i);
        if(PCChar->Def.MagicSpells[i/32] & (1<<(i&31)) &&
            SpellPtr->Name[0] &&
            PCChar->ManaPoints >= SpellPtr->Cost) {
            x = i % 4 * 150;
            y = i / 4 * 20;
            m_Font.Print(m_SpellController.GetSpell(i)->Name,
                        x+8, y+8);
        }
    }
    m_Graphics.EndScene();
}
m_Graphics.Display();

return TRUE;
}
```

## cApp::GetCharacterAt

Одна из самых крутых функций — **GetCharacterAt**, перебирающая список персонажей и определяющая, какой из них находится в указанных экранных координатах. Хотя для этого было бы достаточно выполнять проверку ограничивающего прямоугольника персонажа в экранных координатах, пройдем дальше, и выполним работу по выбору на уровне полигональных граней:

```
long cApp::GetCharacterAt(long XPos, long YPos)
{
    D3DXVECTOR3 vecRay, vecDir;
    D3DXVECTOR3 vecMeshRay, vecMeshDir;
    D3DXVECTOR3 vecTemp;
    D3DXMATRIX matProj, matView, *matWorld;
    D3DXMATRIX matInv;
    DWORD FaceIndex;
    BOOL Hit;
    float u, v, Dist;
    sCharacter *CharPtr;
    sMesh *MeshPtr;

    // Получение родительского объекта персонажа
    if((CharPtr = m_CharController.GetParentCharacter()) == NULL)
        return -1;
}
```

Вы собираетесь проверить каждый персонаж, чтобы увидеть, по какому из них щелкнули, так что **GetCharacterAt** начинает с проверки того, есть ли у нас какие-нибудь персонажи. Теперь вы должны вычислить луч,

исходящий из местоположения указателя мыши, для проверки пересечения с полигонами каждой сетки.

```
// Получаем матрицу проекции, матрицу вида
// и инвертированную матрицу вида
m_Graphics.GetDeviceCOM()->GetTransform(D3DTS_PROJECTION,
                                         &matProj);
m_Graphics.GetDeviceCOM()->GetTransform(D3DTS_VIEW,
                                         &matView);
D3DXMatrixInverse(&matInv, NULL, &matView);

// Вычисляем вектор луча выбора в экранном пространстве
vecTemp.x = (((2.0f * (float)XPos) /
              (float)m_Graphics.GetWidth()) - 1.0f) /
              matProj._11;
vecTemp.y = -(((2.0f * (float)YPos) /
                (float)m_Graphics.GetHeight()) - 1.0f) /
              matProj._22;
vecTemp.z = 1.0f;

// Преобразуем луч в экранном пространстве
vecRay.x = matInv._41;
vecRay.y = matInv._42;
vecRay.z = matInv._43;
vecDir.x = vecTemp.x * matInv._11 +
            vecTemp.y * matInv._21 +
            vecTemp.z * matInv._31;
vecDir.y = vecTemp.x * matInv._12 +
            vecTemp.y * matInv._22 +
            vecTemp.z * matInv._32;
vecDir.z = vecTemp.x * matInv._13 +
            vecTemp.y * matInv._23 +
            vecTemp.z * matInv._33;
```

Теперь луч сконфигурирован (точно так же, как вы конфигурировали его в главе 8, «Создание трехмерного графического движка»), и вы продолжаете исполнение, входя в цикл, перебирающий всех персонажей. Для каждого персонажа функция **GetCharacterAt** сканирует все образующие персонаж сетки, выполняя проверку пересечения луча с полигоном.

```
// Сканируем каждый персонаж и проверяем пересечение
while(CharPtr != NULL) {
    // Сканируем сетки персонажа
    MeshPtr = CharPtr->Object.GetMesh()->GetParentMesh();

    while(MeshPtr != NULL) {
        // Преобразуем луч и направление с помощью
        // матрицы мирового преобразования объекта
        matWorld = CharPtr->Object.GetMatrix();
        D3DXMatrixInverse(&matInv, NULL, matWorld);
        D3DXVec3TransformCoord(&vecMeshRay, &vecRay, &matInv);
        D3DXVec3TransformNormal(&vecMeshDir, &vecDir, &matInv);

        // Проверяем пересечение
        D3DXIntersect(MeshPtr->m_Mesh, &vecMeshRay, &vecMeshDir,
                      &Hit, &FaceIndex, &u, &v, &Dist);
    }
}
```

**ВНИМАНИЕ!**

Будьте внимательны! Если ваши сетки были созданы с указанием флага, делающего их доступными только для чтения, вызов **D3DXIntersect** скорее всего приведет к сбою. Это происходит из-за того, что функция **D3DXIntersect** не может заблокировать и прочитать буфер вершин сетки. Чтобы гарантировать, что вызов **GetCharacterAt** будет работать, убедитесь, что при создании сеток указываете флаг, позволяющий чтение.

Обратите внимание, что для каждого рассматриваемого персонажа вы выполняете смещение координат луча в соответствии с ориентацией персонажа в мире (получаемой из матрицы преобразования объекта персонажа). Вы поступаете так потому что функция проверки пересечения не принимает во внимание мировое преобразование каждого персонажа; это делаете вы.

Если есть пересечение с полигоном, функция возвращает идентификационный номер персонажа. С другой стороны, если пересечения не обнаружено, вы проверяете следующую сетку персонажа, а потом переходите к следующему персонажу в списке. Если больше персонажей не найдено, функция возвращает признак ошибки (−1).

```
// Проверяем, пересекает ли луч персонаж
// и возвращаем ID, если да
if(Hit == TRUE)
    return CharPtr->ID;

// Переходим к следующей сетке
MeshPtr = MeshPtr->m_Next;
}

// Переходим к следующему персонажу
CharPtr = CharPtr->Next;
}

return -1; // Нет попаданий
}
```

## Использование боевых аранжировок

Хотя я намеренно жестко закодировал функции, определяющие персонажей, используемых в боевой последовательности демонстрационной программы Battle, в реальном игровом приложении вы основываете выбор монстров на области карты, в которой сейчас находится игрок.

Например, в заброшенном доме игроки столкнутся с привидениями и зомби. В сельской местности игроки могут столкнуться с обитающими в этой области крупными животными. Вы должны определить, с какими типами монстров (и где) могут встретиться игроки.

Лучший способ определить, какие монстры в какой из боевых последовательностей должны сражаться, — это использование *боевых аранжировок* (*battle arrangements*), заранее сконфигурированных наборов



монстров. В заброшенном доме у вас будет одна аранжировка с двумя призраками и другая с тремя зомби. Когда начинается сражение, игра случайным образом выбирает одну из аранжировок.

Как хранить аранжировки? Конечно, в виде скриптов! В главе 12 вы узнали, как загружать стартовый скрипт. Просто определите, какой скрипт будет загружен, и затем обработайте его, позволив скрипту добавить монстров на арену сражения.

Важно, что вы можете принудительно запустить определенную боевую последовательность. Например, если пришло время сразиться с главным злодеем уровня, вы можете явно загрузить этот скрипт. Внутри скрипта вы можете загрузить песню для воспроизведения, сообщающую, что настало время главной битвы. Когда вы используете скрипты, нет никаких ограничений!

Демонстрационный пример полной игры, названной The Tower, рассматриваемый в главе 16, «Объединяем все вместе в законченную игру», демонстрирует, как использовать боевые аранжировки в реальном игровом приложении. Изучите его для получения дополнительных деталей о применении скриптов в ваших боевых последовательностях (на CD-ROM загляните в папку \BookCode\Chap16\Game).

## **Заканчиваем с боевыми последовательностями**

Если вы похожи на меня, то почувствовали, что боевые последовательности делают ролевые игры интереснее. Что может быть лучше для снятия дневного стресса, чем прорубиться сквозь орды злодеев? Чувство, которое я испытываю, когда мое последнее заклинание высвобождено и очищает поле битвы, всегда поднимает мое настроение.

Когда вы имеете дело с битвами для своей игры, помните, что они должны быть достаточно простыми, чтобы удерживать интерес игрока. Никто не хочет погрязть в бесполезных возможностях или деталях; просто закрутите интересный сюжет и позвольте мчаться вперед.

Хотя разработанный в этой главе проект боевой последовательности относительно прост, он дает хорошее основание, которое вы затем можете расширять и показать каждому, что можете сделать. Попробуйте добавить возможность менять экипировку в ходе боя, или дать персонажам возможность убежать. Изменяйте сетку арены в зависимости от местоположения вашего игрока в мире. Отслеживайте, какие предметы и деньги роняют погибшие монстры и добавляйте их к имуществу игрока, когда битва закончена.

Немного поработав вы превратите демонстрационный проект в полнофункциональный движок боевых последовательностей!

**Программы на CD-ROM**

Каталог \BookCode\Chap14\ содержит следующие программы, демонстрирующие внешние боевые последовательности:

**Battle** — демонстрация трехмерного боевого движка. Щелкните по целевому персонажу, дождитесь появления боевых вариантов, и выберите как атаковать. Местоположение: \BookCode\Chap14\Battle\.



# Глава 15

## Сетевой режим для многопользовательской игры

Хотя уничтожение бесчисленных орд зла и спасение бесчисленных миров достаточно интересно, чтобы отнять у вас сон, спустя некоторое время это занятие становится довольно скучным. Несмотря на все воздвигаемые перед игроком трудности, разработчики не могут обеспечить интригу и сложность человеческого разума, которую игроки хотят видеть в своих виртуальных противниках.

Войдите в мир сетевых игр, где настоящие противники с настоящими мыслями и реакциями ожидают вашего участия в игровых действиях, что изменяет обычный игровой процесс. Вы больше не будете сидеть, избивая безмозглых приспешников. Теперь вы столкнетесь с сотнями хитроумных игроков, желающих либо присоединиться к вам, либо встать на вашем пути.

Ваша игра должна дать игрокам возможность подключиться к Интернету и найти других игроков, желающих объединить силы и устроить разборки в истинно многопользовательском стиле.

Данная глава является вашим руководством для выполнения этого подвига. В главе вы узнаете следующее:

- Основы многопользовательских игр.
- Как спроектировать архитектуру многопользовательской игры.
- Пример проекта многопользовательской игры.
- Как создать игровой сервер и клиентское приложение.

### Безумная многопользовательская сеча

В клубках дыма и вспышках света мой герой материализуется позади небольшого здания, находящегося в маленьком городке. Завернув за угол, он видит большую часовню. Исходящее от этой проклятой часовни зловоние смерти заполняет воздух. Цель героя ясна — войти в этот храм зла и удалить проклятие из его глубин.

Слева стоят две зловещие фигуры, все в шрамах от бесчисленных сражений. Приблизившись к мрачному дуэту герой понимает, что это его

товарищи по оружию, готовые присоединиться к нему в исследовании мрачных глубин. После быстрой прогулки к местному магазину для приобретения некоторых целебных снадобий, они углубляются в часовню.

Орды злобных созданий всевозможных размеров немедленно ощущают присутствие искателей приключений. Существа собираются избавить свой мир от незваных гостей. Но у искателей приключений другие планы, и, благодаря своей силе и хитрости, они побеждают всех монстров, встающих на их пути. Цель близка и они чувствуют, что смогут достичь ее.

Как видно из приведенного сценария, у моих приятелей и меня было почти идеальное приключение (спасибо игре Diablo от Blizzard Entertainment). Я говорю почти, потому что победоносно прошествовав сквозь толпы управляемых компьютером демонов, мы не выстояли против вновь прибывшего — таинственного человека только что подключившегося к игре. Магически перенесясь к нашему текущему местоположению он начал делать фарш из нас.

Лежа на земле я (герой) смотрел, как грабитель перебирает мое имущество, забирая себе сокровища, добытые мной с таким трудом. Как кто-то может быть таким порочным, таким вероломным — и таким сильным? Очевидно, этот парень был здесь гораздо дольше, чем я; его мощь потрясает, и я поклялся достичь такого же уровня силы, чтобы я мог рисковать дальше и показать кто здесь главный!

Покинув текущее приключение я снова подключился к лобби-серверу Diablo и начал новую игру. Однако на этот раз я отправился один. С каждым убийством я становился сильнее; каждое найденное сокровище я вкладывал в покупку лучшего оружия. Скоро я буду готов выследить дикаря, который ранее ограбил моих приятелей и меня.

Это одно из множества приключений, которые мне посчастливилось испытать. Те из вас, кто «потерял» (то есть *вложил*) драгоценные часы играя в Diablo, могут подтвердить насколько интересной может быть игра. Сетевые возможности в Diablo делают игру более ценной и вы можете захотеть продублировать их в вашем проекте. Итак, после такой рекламы вы готовы перейти к действиям.

## Проектирование многопользовательской игры

Сетевые игры дали игрокам абсолютно новый способ состязаться с другими людьми, и, как показал рассказ в предыдущем разделе, эти столкновения могут быть забавными или бросать вызов ловкости других игроков.

Для многопользовательских игр потребуется немного больше усилий при проектировании. В однопользовательской игре игрок берет под свой контроль героя игры и таким образом взваливает на себя спасение мира. В многопользовательской игре могут быть тысячи других игроков, и каждый желает стать единственным истинным героем, что, очевидно, невозможно.

При переходе от однопользовательских к многопользовательским играм происходит смена цели. Рассмотрим, например, игры подобные Ultima Online от Origin. Если вы играли в такую игру, то понимаете, что у нее нет никакой реальной цели. Здесь нет ни главного злодея, над которым нужно одержать победу, ни ведущего вас за руку сценария, ни ощущения, что единственный человек может изменить мир.

Так почему же кто-то играет в Ultima Online? По одной причине — это доставляет удовольствие. Вы больше не будете самым сильным во вселенной, поскольку к вам присоединяются тысячи других игроков с теми же самыми мыслями о славе. Ultima Online дает людям возможность объединять силы для сражения с бесконечными ордами зла, выполнения небольших заданий или даже сражений друг с другом.

С человеческим интеллектом и мощностью игрового движка Ultima Online, игроки могут достигать придуманных ими самими целей, что делает приобретение опыта похожим на реальную жизнь. Хотя Ultima Online и не предоставляет сценария (а также возможности выиграть в игре), она продолжает оставаться флагманом сетевых игр.

С другой стороны, можно взглянуть на такую игру, как Phantasy Star Online от Sega, которая вынуждает игроков объединять силы в борьбе с превосходящими ордами зла. В Phantasy Star Online есть и сюжет (хотя он не слишком конкретен и кажется лишь добавлением). И самое замечательное, что в Phantasy Star Online вы можете выиграть. Верно, в каждой новом сеансе игры история начинается заново, ожидая ваших персонажей, готовых прорубиться сквозь тысячи монстров, чтобы убить их главного предводителя.

Отличия между Ultima Online и Phantasy Star Online существенны, и все же обе объединяют тысячи игроков. Что же доставляет наибольшее удовольствие игрокам, и какие возможности они ожидают от сетевых игр? Это трудные вопросы, но давайте взглянем, что предлагает каждая из игр, и отметим те возможности, которые хочется включить в собственный игровой проект.

- **Развитие персонажа.** Зачем напрасно тратить время на игру, если ваши усилия никак не отражаются? Причина, по которой вы прорубаетесь через мировые орды бестий в том, что ваш персонаж становится «злее и здоровее» и продолжает идти к дальнейшим достижениям.
- **Развивающийся мир.** Никогда не меняющийся мир примитивен; после исследования он, несмотря ни на что, остается все таким же. Популярные игры позволяют менять мир с помощью новых уровней для исследования или новых подвернувшихся заданий.
- **Объединение и противостояние игроков.** Людям необходимо взаимодействовать с другими людьми — такова жизнь. Имея в распоряжении Интернет, вам требуется только возможность

присоединения игроков к многопользовательским действиям, даже если эти игроки находятся в состоянии войны друг с другом.

- **Новости, секреты и крутые предметы для открытий.** Что хорошего в последовательном исследовании мира, если по пути нельзя прихватить пару вещей — более сильное оружие, страшную броню и, может быть, сверхсекретный магический предмет, который действительно обратит ход битвы.
- **Настоящая сюжетная линия.** Проходит ли сюжетная линия через всю игру, или присутствует только в отдельных заданиях в игре, хороший сценарий придает игре дополнительный шик.
- **Возможность выиграть.** Такие игры, как Phantasy Star Online и Diablo, дают игроку реальную возможность паобедного финала. Конечно, обычно игра может продолжаться многие часы, но в ней остается возможность победить последнего великого злодея и спасти мир.

Помните, что независимо от того какие возможности вы включите в игру, ваша цель — создать интересную игру, в которую будут играть снова и снова. Посмотрите на имеющиеся на рынке игры, чтобы определить в каком направлении двигаться, и попытайтесь сделать ваш проект увлекательным.

Чтобы помочь вам создать многопользовательскую игру, я написал демонстрационную программу, названную Network Game. В следующем разделе вы узнаете, как я проектировал игру, а оставшаяся часть главы будет посвящена особенностям программирования игры.

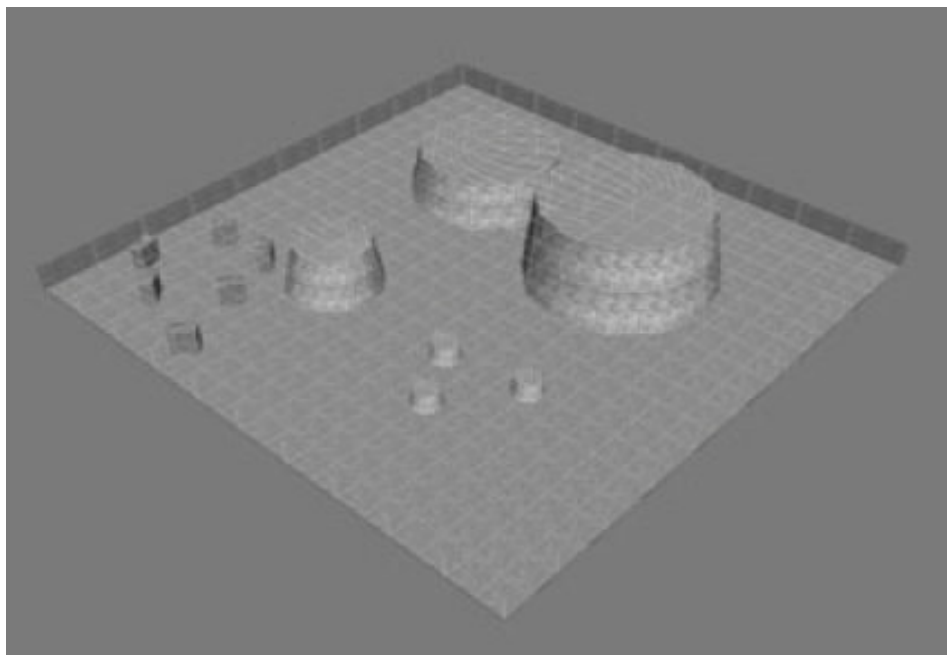
## Демонстрационная программа Network Game

Демонстрационная программа Network Game находится на прилагаемом к книге CD-ROM (загляните в каталог \BookCode\Chap15) и является основой проекта. В игре есть единственный большой уровень, игроки могут присоединяться к игре и начать перемещаться и атаковать друг друга (хотя никаких повреждений нет и никто не умирает).

Игровой уровень — это большая сетка (как показано на рис. 15.1), использующая для визуализации объект класса **NodeTree** (об использовании класса **NodeTree** подробно рассказывалось в главе 8, «Создание трехмерного графического движка»).

Игрок представлен единственной сеткой (еще одна сетка представляет его оружие). Взгляните на рис. 15.2, показывающий использование сеток игрока и его оружия.

Когда запущены механизмы демонстрационной игры, игрокам позволено присоединиться к идущей игре и начать перемещаться по уровню. Локально игрок использует сетки для визуализации трехмерного вида игрового мира. Точка просмотра каждого игрока расположена немного выше и позади его персонажа, как показано на рис. 15.3.



*Рис. 15.1. Уровень программы Network Game. Несколько препятствий позволяют игрокам скрываться друг от друга*



*Рис. 15.2. Полностью анимированная трехмерная модель (слева), несущая оружие (справа) представляет игровой персонаж*





*Рис. 15.3. Демонстрационная игра показывает привлекательный трехмерный ландшафт и анимированных виртуальных игроков*

Для перемещения персонажа игрок использует клавиши управления курсором; нажатие клавиши перемещает игрока в соответствующем направлении относительно камеры. Перемещение мыши вправо и влево меняет направление камеры, давая игроку 360-градусный обзор. Игроки могут атаковать друг друга нажимая пробел, в результате их персонаж взмахивает своим мечом, надеясь поразить другого персонажа.

Хотя демонстрационная игра очень проста, она дает вам основы, необходимые чтобы начать создавать свою игру. Помните, что все дороги в конце концов сольются в одну — получение готовой многопользовательской игры, и для этого во-первых вы должны создать лежащую в основе архитектуру игры.

## Создание архитектуры многопользовательской игры

Если вы создаете игру и останавливаетесь посередине, пытаясь добавить многопользовательские возможности (которые вы не планировали включать), велика вероятность того, что вы столкнетесь с трудностями, пытаясь заставить все работать правильно. Помните — подготавливаться надо заранее, и если ваша игра собирается быть совместимой с многопользовательскими возможностями, вы должны убедиться, что у вас есть прочное основание для последующей работы.

Начните с понимания того, чему вам предстоит противостоять, используя сеть, и что вы можете сделать, чтобы обеспечить правильную работу. Будет полезно посмотреть как используется сетевая архитектура клиент/сервер. В главе 5, «Работа с сетью с DirectPlay», объяснялись основы работы с сетевыми коммуникациями на стороне клиента и сервера, а сейчас пришло время посмотреть как сервер и клиент могут эффективно работать вместе с точки зрения игрового процесса.

## Работаем вместе: клиент и сервер

Клиент и сервер непрерывно общаются между собой. Когда игрок (клиент) выполняет в игре какое-либо действие, информация об этом действии отправляется в виде сообщения серверу для проверки. Сервер, обеспечивая поддержку синхронизации, получает действия игроков, обновляет игровой мир и рассылает обновленное состояние игры клиентам. Таким образом, сервер управляет всем игровым миром, а клиенты являются простыми системами для сбора действий игроков (и отображения их результатов на экранах клиентов).

Типов сообщений, пересылаемых между сервером и клиентом, может быть очень много, но в хорошо продуманном проекте эти сообщения будет легко обслуживать. Сообщения могут быть следующими:

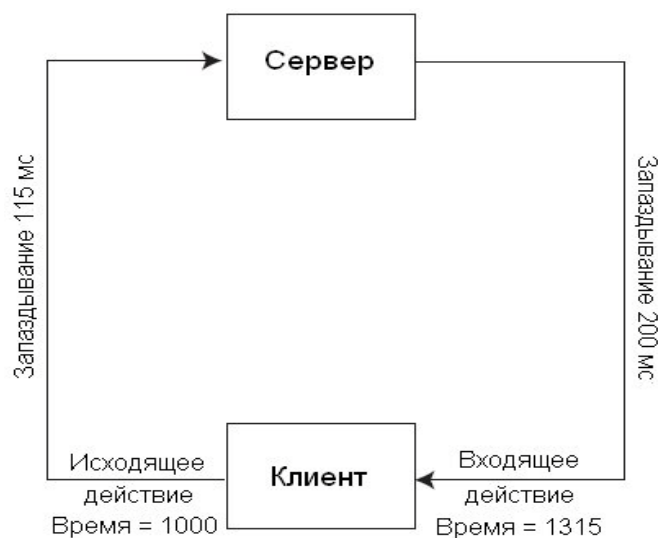
- **Запрос подключения.** Присоединение к игре означает подключение к серверу. Однако, присоединиться может не каждый; к серверу уже может быть подключено максимально возможное количество игроков, или у игрока может не быть допустимой учетной записи. Как только клиент подключился, начинается настоящее действие!
- **Навигация.** Игрок может перемещаться по картам, обычно используя клавиши управления курсором на клавиатуре или щелкая по требуемому месту на карте. Клиенты отсылают свои запросы на перемещение и ждут, пока сервер не вернет им сообщения об обновлении игрового состояния.
- **Сражения.** Размахивание мечами и разбрасывание заклинаний, кажется, со стольким многим приходится иметь дело. Но если очистить все от шелухи, вы обнаружите, что битва это только лишь атакующий с его способом атаки и обороняющийся с его способом защиты. Клиенты только формируют запросы на сражения; работа сервера — получить запрос на сражение и преобразовать его в изменения игрового состояния.
- **Управление ресурсами.** В мире, полном вещей, игроки хотят иметь возможность купить, продать, найти или использовать любой из ресурсов, до которых дотянутся их руки. Я начинаю повторяться, поскольку управление ресурсами исходит от клиента, и запросы отправляются серверу, чтобы быть использованными для обновлений.

- **Общение.** Что за удовольствие от многопользовательской игры без взаимодействия с другими игроками? Персонажи говорят друг с другом, чтобы узнать важную информацию, или просто потрепаться. В любом случае, это вопрос отображения нескольких строчек текста. Это общение работает в обоих направлениях, клиенты отправляют текст серверу, а сервер возвращает им текст, который должен быть показан.
- **Обновление игрового состояния.** Как упоминалось ранее, сервер должен предоставить всем клиентам возможность периодически узнавать состояние игры, и сообщение об изменении игрового состояния это просто билет. Сообщение об изменении игрового состояния обычно включает местоположение всех игровых персонажей, плюс информацию о предметах и других игровых ресурсах.

При использовании упомянутой выше сетевой архитектуры на ум сразу приходит пара вещей. Во-первых, поскольку только сервер ответственен за управление состоянием игры, все подключенные клиенты должны ожидать упомянутых периодических обновлений для поддержания хода игры.

К сожалению, скорость передачи данных по сети не позволяет осуществлять мгновенную передачу, поэтому данные, проходящие от клиента к серверу и обратно, задерживаются. Эта задержка в процессе передачи называется *запаздыванием (latency)*, и это время ожидания может вызвать проблемы в вашей игре.

Поскольку только серверу позволено делать изменения в игровом мире, сервер должен подтверждать допустимость происходящих действий пользователей. Как видно на рис. 15.4, игроки сталкиваются с проблемой из-за задержки, между временем, когда действие было инициировано, и временем, когда оно происходит. Эта задержка действий называется *отставанием (lag)* и может сделать игровой процесс неустойчивым (и непригодным для игры).



**Рис. 15.4.** Клиент отправляет сообщение и ждет ответа от сервера, ощущая задержку передачи (отставание)

Чтобы обеспечить плавность событий и помочь смягчить эффекты задержки и отставания, клиентам позволяется производить незначительные изменения в игровом мире между поступающими с сервера обновлениями. Эти небольшие изменения обычно являются только обновлениями перемещения персонажа. В этом случае клиент не ждет обновлений от сервера, чтобы переместить персонажи; клиент может только предположить, как обновлять все персонажи, основываясь на их последнем известном состоянии (рис. 15.5). Подобные предположения называются *расчетом траектории* (*dead reckoning*) и применяются в сетевых играх.

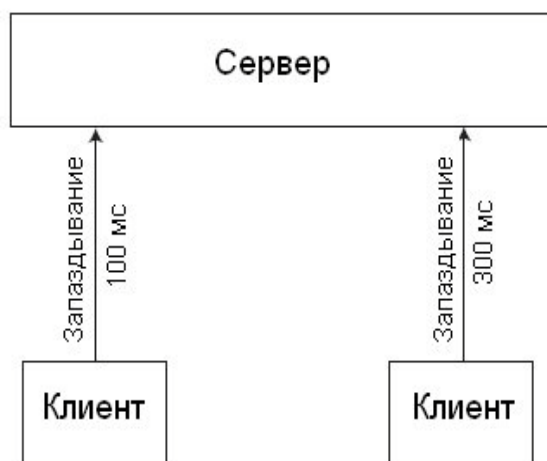


**Рис. 15.5.** Благодаря расчету траектории, идущий персонаж продолжает движение, пока сервер не сообщит, что персонаж остановился

В более сложных действиях, таких как сражения, использование расчета траектории неприменимо. Сервер является верховной властью, и если системе необходимо определить кто, кого и как сильно ударил, она должна отправить запрос серверу.

Как упоминалось, вторая проблема при использовании сетевых систем — это отсчет игрового времени. Взгляните: попытка синхронизировать десятки клиентов практически нереальна. Каждый компьютер подключен к сети с различным временем запаздывания; некоторые клиенты дольше отправляют сообщения серверу и позже принимают их обратно от сервера.

На стороне клиента один игрок может выполнить перемещение точно в тот же момент, что и другой, но поскольку этим действиям требуется время, чтобы достичь сервера, у клиента с более быстрым соединением есть преимущество (рис. 15.6).



*Рис. 15.6. У клиента слева задержка меньше, чем у игрока справа, поэтому его действия первыми достигнут сервера и будут обработаны раньше*

Все сообщения, полученные клиентом и сервером, записываются вместе со временем их получения. Сервер использует это время, чтобы определить, как обновлять состояние игроков. Например, если полученное сервером сообщение не обрабатывалось в течение 100 мс, сервер компенсирует пропущенное время в ходе выполнения обновления. Тоже самое верно и для клиента. Если действие должно быть обновлено (особенно при использовании расчета траектории), это время (время, прошедшее с момента получения сообщения) используется для соответствующего перемещения персонажей.

### **ВНИМАНИЕ!**

Если вы оставляете важные решения (такие, как сражения) на стороне клиента, вас ждут неприятности, поскольку взломщики и мошенники будут использовать эти лазейки для получения преимуществ. Помните, что только сервер ответственен за отслеживание хода игры; клиенты — это просто порталы в игровой мир.

Теперь, рассмотрев совместную работу клиентов и сервера, пристальнее взглянем на каждую из этих систем.

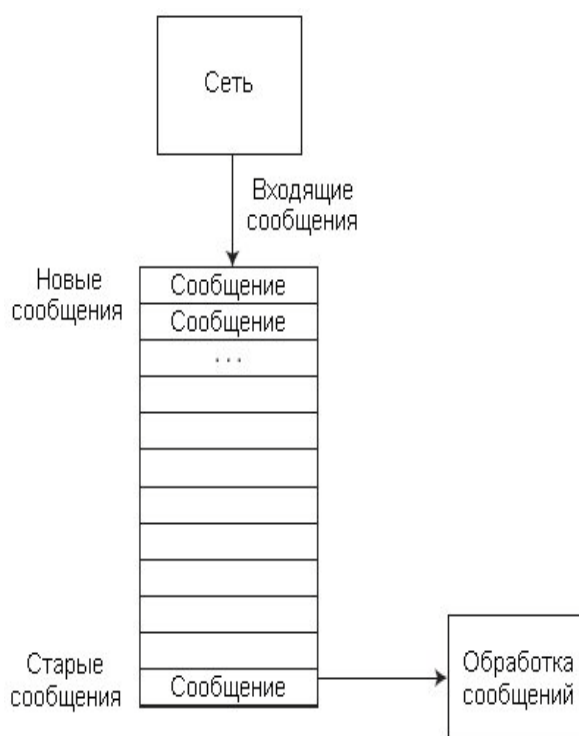
## **Взгляд на сервер**

Игровой сервер — это специализированная часть программного обеспечения. Ему не нужна захватывающая графика, воспроизведение мелодий или даже выделенные функции ввода. Серверу необходимо только обрабатывать получаемые от подключенных игроков действия и также часто отправлять обновления клиентам.

После запуска сервер попадает в цикл, непрерывно обрабатывая входящие сетевые сообщения, обновляя состояние всех подключенных игроков, основываясь на их последнем известном действии, и рассылая обновления.

Сервер получает несколько сетевых сообщений — запрос подключения, уведомление об отключении и действия игрока. Действия игрока целиком зависят от проекта игры, и в демонстрационной программе из этой главы включают только ходьбу игрока, остановку и атаку оружием.

Полученное от клиента сетевое сообщение помещается в *очередь сообщений (message queue)*. Использование очереди сообщений ускоряет сетевые операции и оставляет больше времени для работы основного приложения (а не потока с кодом работы с сетью). Сервер управляет очередью сообщений (стеком сообщений), хранящей все входящие сообщения. Поступающее сообщение добавляется в очередь. Сервер постоянно извлекает наиболее старые сообщения и перенаправляет их различным функциям для обработки. Этот процесс обработки сообщений показан на рис. 15.7.



**Рис. 15.7.** Сетевые сообщения вставляются в очередь в порядке их поступления. Очередь сообщений гарантирует, что следующим сервером будет обработано самое старое из сообщений

Имея дело с запросами на подключение игроков, сервер сначала проверяет, есть ли открытый слот для игрока. Если да, от клиента запрашиваются данные игрока, которые сохраняются в локальной структуре. Все присутствующие в игре игроки уведомляются, что к драке присоединился новый игрок, и игра продолжается. Слот освобождается когда игрок отключается.

**COBET**

Для улучшения синхронизации клиент и сервер вычисляют временную задержку получения сообщения. Вы увидите как вычислять задержку в разделе «Вычисление запаздывания» далее в этой главе.

Вскоре приходится иметь дело с действиями игрока; все действия игрока просто меняют состояние игрока. В рассматриваемом примере используются только состояния ходьбы, остановки, атаки и получения повреждений. Эти состояния используются в каждом кадре для обновления информации игрока. Получив действие игрока сервер отправляет его всем другим подключенным игрокам, чтобы они могли обновлять их игровые состояния (разумеется, между обновлениями сервера).



Кроме работы с сетевыми сообщениями, сервер обновляет состояние игроков. Если последним известным состоянием персонажа игрока была ходьба в заданном направлении, игрок продолжает движение в этом направлении. Сервер, как ответственный за все, должен выполнять проверку столкновений, чтобы перемещающиеся персонажи не ходили сквозь стены!

Для каждого имеющегося в игре действия и состояния вы добавляете к серверу логику для обработки персонажей. Например, состояние атаки требует, чтобы сервер отклонял последующие изменения состояния от игрока, пока состояние атаки не будет сброшено (через одну секунду). Когда клиент инициирует нападение, сервер вычислит, какие другие клиенты были поражены и уровень нанесенных им повреждений.

Реализация сервера проста. После того, как вы создали основу для работы, можно легко добавлять к серверу дополнительные возможности. Помимо добавления новых действий, которые могут выполнять игроки, вы можете добавить такие возможности, как управление учетными записями игроков. Однако, пришло время взглянуть на имеющиеся вещи со стороны клиента.

## **Взгляд на клиента**

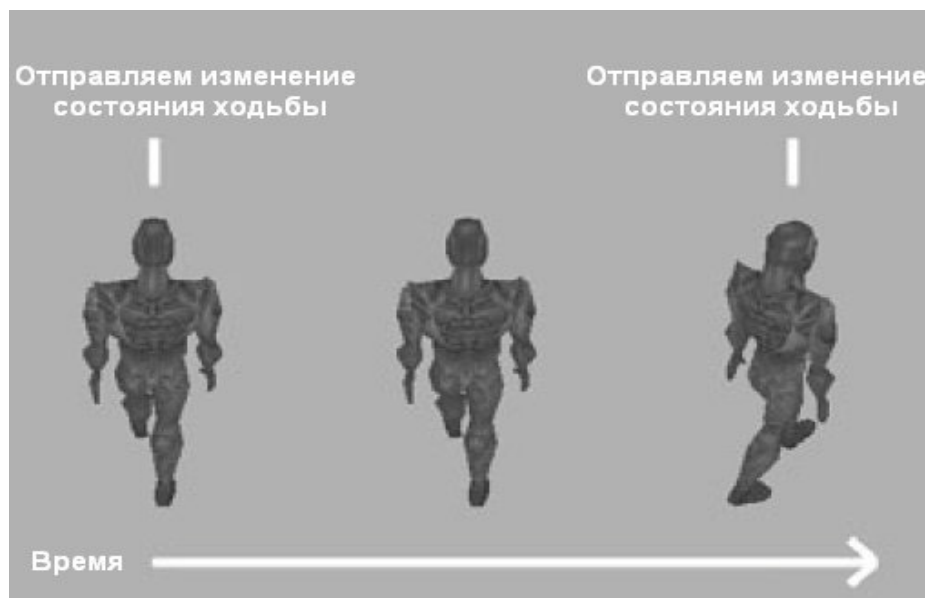
Установив соединение клиенту необходимо собрать информацию об управлении локальным игроком и отправить ее на сервер. В промежутках между получениями обновлений от сервера клиент предполагает (используя отслеживание траектории) как следует управлять всеми игровыми персонажами, основываясь на их последнем известном состоянии.

Например, все персонажи, шедшие при последнем обновлении, продолжают идти, пока сервер не сигнализирует, что их нужно остановить. Благодаря этому игровой процесс получается плавным, и с хорошим сетевым соединением обновления от сервера поступают достаточно часто, чтобы игра оставалась полностью синхронизированной.

Как показано на рис. 15.8, когда клиент выполняет изменение действия (такое, как ходьба в направлении, отличном от заданного в последнем известном состоянии), это изменение состояния немедленно передается серверу, который сразу же рассылает его всем подключенным клиентам. Благодаря этому синхронизация становится еще лучше.

Если говорить об изменении действий игрока, какие точно действия может выполнять игрок? Например, перемещение. Когда игрок ходит по карте, направление его движения передается на сервер. Заметьте, что отправляется только направление движения.

Если вы позволите клиентам указывать координаты, когда они перемещаются, у мошенников возникнет искушение подменить эти значения. Вместо этого сервер модифицирует координаты игрока и отправляет эти координаты обратно клиенту (в этот раз не имеет значения, модифицирует ли мошенник эти значения, поскольку они не оказывают влияния на сервер).



**Рис. 15.8.** Клиент отправляет изменение состояния серверу только когда игрок выбирает перемещение в направлении отличном от предыдущего, что экономит время, поскольку не приходится пересылать одно и то же направление перемещения снова и снова

Для определенных действий, таких как ходьба, клиентам разрешается изменять свое собственное состояние. В результате игроки могут перемещаться в промежутках между получением обновлений от сервера. Для таких действий как атака, изменение состояния только отправляется на сервер, который, в свою очередь, обрабатывает атаку и рассылает соответствующие изменения состояний всем клиентам.

Игроки могут выполнять обновления только каждые 33 мс. Это временное ограничение обновлений введено для того, чтобы клиенты не затопили сервер тысячами действий. Сохраняя минимальный уровень действий сервер может обрабатывать все более быстро и игровой процесс остается плавным.

Всякий раз, когда сервер отправляет клиенту критически важные обновления, клиент немедленно меняет состояние рассматриваемого персонажа или персонажей (здесь не требуется очередь сообщений). Это обновление может также затрагивать локального игрока, поскольку вы перемещаетесь вокруг и могло произойти несколько переключений действий из-за синхронизации клиента с сервером.

Ладно, достаточно исследований; перейдем к созданию настоящей сетевой игры!

## Работа над игровым сервером

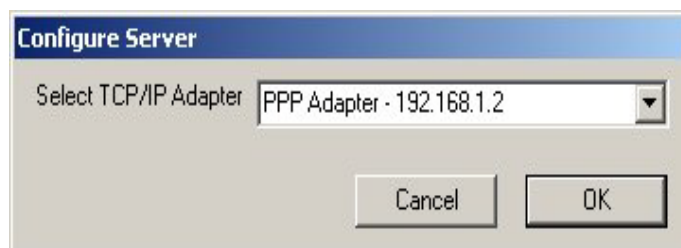
Вы уже читали о том, насколько простым может быть сервер. Чтобы претворить теорию в практику (и помочь вам создать собственные многопользовательские игры) я создал серверное приложение, которое вы найдете на прилагаемом к книге CD-ROM (в папке \BookCode\Chap15\Server). В этом разделе вы узнаете как разработать



лежащую в основе архитектуру сервера сетевой игры и создать серверное приложение.

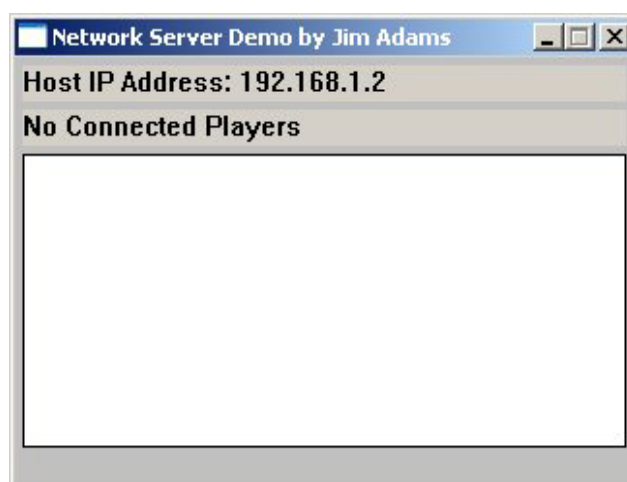
Игровой сервер по своей природе главенствующий. Для обработки графики, работы с сетью и ввода вы можете использовать игровое ядро. Поскольку в главе 6, «Создаем ядро игры», рассказывалось как использовать игровое ядро, я пропущу формальности и перейду к сути. Чтобы использовать серверное приложение следуйте инструкции:

1. Найдите приложение **Server** на прилагаемом к книге CD-ROM. Когда вы запустите приложение, появится диалоговое окно **Configure Server** (рис. 15.9).



*Рис. 15.9. Диалоговое окно **Configure Server**, появляющееся сразу после запуска приложения сервера, позволяет выбрать TCP/IP-адаптер, который будет использоваться на главном узле игры*

2. В диалоговом окне **Configure Server** выберите TCP/IP-адаптер, который будет использоваться на главном узле игры.
3. Щелкните **OK** для запуска игры. Откроется окно **Network Server Demo** (рис. 15.10). В этом окне отображается IP-адрес сервера, количество игроков и список подключенных к серверу игроков (если такие есть).



*Рис. 15.10. Окно **Network Server Demo** отображает IP-адрес главного узла, количество подключенных игроков и имя каждого игрока*

4. Теперь игроки могут подключаться к серверу и играть в игру. Одновременно к игре может быть подключено только восемь игроков, но вы можете увеличить их количество в исходном коде.
5. Для закрытия сервера (и отключения всех игроков) нажмите **Alt+F4**.

Поэкспериментировав с сервером двигайтесь дальше и загрузите исходный код.

Настоящая работа происходит за кулисами, где обрабатываются сообщения, перемещаются персонажи и происходит управление всем игровым миром. Приложение использует наследуемый от **cApplication** класс с именем **cApp**; части класса серверного приложения вы увидите в этом разделе.

Пропустив стандартные функции инициализации приложения (такие, как инициализация графики и системы ввода), пройдемся шаг за шагом по функциональности сервера, начав с поддержки игроков.

## Хранение информации игрока

Игрокам в игре разрешено только бродить вокруг и размахивать своим оружием (задевая других игроков). Сервер отслеживает текущее состояние каждого игрока (ходьба, остановка, взмах оружием или получение повреждений), координаты в мире, направление в котором обращен игрок и скорость его ходьбы (если он идет).

Эти данные игрока хранятся внутри структуры с именем **sPlayer**. Поскольку у каждого подключенного игрока есть его собственный уникальный набор данных, для хранения информации выделяется массив структур **sPlayer**. Количество выделяемых структур с данными игроков и максимальное количество подключенных к игровой сессии игроков хранится в макросе **MAX\_PLAYERS** и изначально установлено равным 8.

Структура **sPlayer** выглядит так (с вспомогательными макросами задания состояний):

```
// Состояния игрока
#define STATE_IDLE 1
#define STATE_MOVE 2
#define STATE_SWING 3
#define STATE_HURT 4

typedef struct sPlayer {
    BOOL    Connected;    // TRUE если игрок подключен

    char    Name[256];    // Имя игрока
    DPNID   dpnidPlayer;  // DirectPlay ID игрока

    long    State;        // Последнее известное состояние (STATE_*)
    long    Time;         // Время последнего обновления состояния
    long    Latency;      // Половина общего запаздывания в мс

    float   XPos, YPos, ZPos; // Координаты игрока
    float   Direction;      // Угол направления взгляда
    float   Speed;          // Скорость перемещения

    // Конструктор
    sPlayer() { Connected = FALSE; Latency = 0; }
} sPlayer;
```

Структура **sPlayer** не так уж велика; у вас есть флаг подключения игрока, имя игрока, идентификатор игрока из **DirectPlay**, текущее состояние игрока (задаваемое с помощью макроопределений состояний), время последнего изменения состояния, значение сетевого запаздывания, координаты игрока, направление и скорость перемещения.

Переменные в **sPlayer** самодокументируемы, за исключением **Latency**. Вспомните, что запаздывание — это задержка, являющаяся результатом передачи данных по сети. Сохраняя время, требуемое сообщению чтобы дойти от сервера до клиента (и наоборот), мы повышаем синхронизированность зависящих от времени вычислений между сервером и клиентом.

Раз речь зашла о зависящих от времени вычислениях, именно для этого предназначена переменная **Time**. Всякий раз, когда сервер обновляет данные игроков, он должен знать время, которое прошло между обновлениями. Каждый раз, когда меняется состояние игрока (по запросу от клиента), в переменной **Time** сохраняется текущее время (минус время запаздывания).

Время также используется для управления действиями. Если игрок взмахивает мечом, сервер прекращает принимать от клиента последующие изменения состояния, пока состояние взмаха мечом не будет очищено. Как происходит очистка состояния? После того, как пройдет заданный промежуток времени! Через одну секунду цикл обновления игрока очищает состояние игрока, переводя его в режим ожидания, что позволяет клиенту снова отправлять сообщения изменения состояния.

Что касается сути передаваемых сообщений, посмотрим как сервер работает с входящими сетевыми сообщениями.

## Обработка сообщений

Вы уже видели сообщения **DirectPlay** в действии, а сейчас сосредоточимся на сообщениях об игровых действиях (изменении состояния). Поскольку у **DirectPlay** есть только три функции, которые интересны для обработки входящих сетевых сообщений (**CreatePlayer**, **DestroyPlayer** и **Receive**), серверу необходимо преобразовать входящие сетевые сообщения в сообщения более соответствующие игровому процессу.

Сервер получает сообщения от клиента через сетевую функцию **DirectPlay Receive**. Эти сообщения сохраняются в буфере **pReceiveData**, находящемся внутри структуры **DPNMSG\_RECEIVE**, передаваемой функции **Receive**. Для буфера выполняется преобразование типа к более удобному для применения в игре виду сообщения, помещаемому в очередь сообщений игры.

Код игрового сервера не работает непосредственно с сетевыми сообщениями. Они обрабатываются небольшим подмножеством функций, получающих входящие сообщения и преобразующих их в игровые

сообщения (помещаемые в очередь сообщений). Код игрового сервера работает именно с этими игровыми сообщениями.

Поскольку может быть много различных типов игровых сообщений, необходима обобщенная структура сообщения, являющаяся контейнером. Каждое сообщение начинается с заголовка, хранящего тип сообщения, общий размер данных сообщения (в байтах) включая заголовок, и идентификационный номер игрока DirectPlay, обычно устанавливаемый отправившим сообщение игроком.

Я взял на себя смелость выделить заголовок в отдельную структуру, что позволит повторно использовать заголовок в каждом игровом сообщении.

```
// Структура заголовка сообщения, используемая во всех сообщениях
typedef struct {
    long Type;          // Тип сообщения (MSG_*)
    long Size;          // Размер передаваемых данных
    DPNID PlayerID;     // Игрок, выполняющий действие
} sMessageHeader;
```

Поскольку может быть много различных игровых сообщений, вы, во-первых, нуждаетесь в общем контейнере, который может хранить все различные игровые сообщения. Этот общий контейнер сообщений является следующей структурой:

```
// Структура сообщения из очереди сообщений
typedef struct {
    sMessageHeader Header; // Заголовок сообщения
    char Data[512];       // Данные сообщения
} sMessage;
```

Элементарно, правда? Структуре **sMessage** необходимо хранить только заголовок сообщения и массив байтов, используемых для хранения зависящих от сообщения данных. Чтобы использовать конкретное сообщение вы должны выполнить приведение типа структуры **sMessage** к другой структуре для доступа к данным.

Например, вот структура, представляющая сообщение об изменении состояния:

```
// Сообщение об изменении состояния
typedef struct {
    sMessageHeader Header; // Заголовок сообщения
    long State;            // Состояние (STATE_*)
    float XPos, YPos, ZPos; // Координаты игрока
    float Direction;       // Направление взгляда игрока
    float Speed;           // Скорость ходьбы игрока
    long Latency;          // Значение запаздывания соединения
} sStateChangeMessage;
```

Для приведения типа структуры **sMessage**, содержащей сообщение об изменении состояния к пригодной для использования структуре **sStateChangeMessage** вы можете использовать следующий фрагмент кода:

```
sMessage Msg;           // Сообщение, содержащее данные

sStateChangeMessage *scm = (sStateChangeMessage*)Msg;

// Доступ к данным сообщения об изменении состояния
scm->State = STATE_IDLE;
scm->Direction = 1.57f;
```

Помимо сообщения об изменении состояния в прототипе сетевой игры используются следующие структуры сообщений:

```
// Сообщение о создании игрока
typedef struct {
    sMessageHeader Header; // Заголовок сообщения
    float XPos, YPos, ZPos; // Координаты создания игрока
    float Direction;       // Направление игрока
} sCreatePlayerMessage;

// Сообщение запроса информации игрока
typedef struct {
    sMessageHeader Header; // Заголовок сообщения
    DPNID PlayerID;       // Какой игрок запрашивается
} sRequestPlayerInfoMessage;

// Сообщение об уничтожении игрока
typedef struct {
    sMessageHeader Header; // Заголовок сообщения
} sDestroyPlayerMessage;
```

У каждого сообщения также есть связанное макроопределение, используемое и клиентом и сервером. Эти макросы сообщений являются значениями, хранящимися в переменной **sMessageHeader::Type**. Макроопределения типов сообщений следующие:

```
// Типы сообщений
#define MSG_CREATE_PLAYER 1
#define MSG_PLAYER_INFO 2
#define MSG_DESTROY_PLAYER 3
#define MSG_STATE_CHANGE 4
```

Вы увидите каждое сообщение в действии в разделах «Обработка игровых сообщений» и «Работа над игровым клиентом», позднее в этой главе, но сейчас проверим как сервер управляет этими относящимися к игре сообщениями.

### ***От сообщений DirectPlay к игровым сообщениям***

Как я упоминал ранее, серверу необходимо преобразовывать сетевые сообщения DirectPlay в относящиеся к игре сообщения, о которых вы только что прочитали. Вы достигаете этого обрабатывая входящие сообщения *подключения* игрока, *отключения* и *получения* данных из DirectPlay и преобразуя их в игровые сообщения.

Чтобы осуществить это преобразование сообщений вы наследуете класс от **cNetworkServer** и переопределяете функции **CreatePlayer**, **DestroyPlayer** и **Receive**:

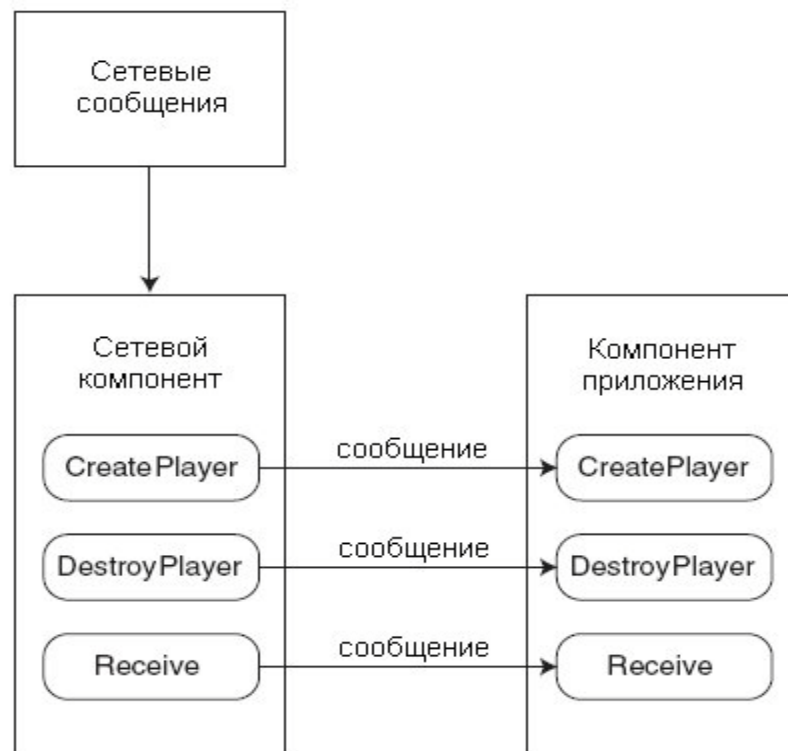
```

class cServer : public cNetworkServer
{
private:
    BOOL CreatePlayer(DPNMSG_CREATE_PLAYER *Msg);
    BOOL DestroyPlayer(DPNMSG_DESTROY_PLAYER *Msg);
    BOOL Receive(DPNMSG_RECEIVE *Msg);
};

```

Поскольку для обработки сообщений в приложении я использую системное ядро, при работе с сетью быстро вырисовывается проблема. Сетевой компонент и компонент приложения — это две отдельные сущности, а это означает, что никакому компоненту не позволено модифицировать закрытые данные другого.

Как показано на рис. 15.11, сетевому компоненту необходим способ перекачать входящие сообщения в приложение, чтобы их можно было обработать, создав три открытые функции, соответствующие функциям сетевого класса.



**Рис. 15.11.** Сетевой компонент пересылает входящие сообщения из переопределенных функций **CreatePlayer**, **DestroyPlayer** и **Receive** в соответствующие открытые функции компонента приложения

Чтобы использовать эти три функции сообщений в компоненте приложения вы конструируете наследуемый от **cApplication** класс, содержащий следующие три открытые функции:

```

class cApp : public cApplication
{
    // Предыдущие данные и функции cApp

private:
    cServer m_Server; // Включаем наследуемый класс сетевого сервера

```

```
public:
    // Функции для перекачки сетевых сообщений в приложение
    BOOL CreatePlayer(DPNMSG_CREATE_PLAYER *Msg);
    BOOL DestroyPlayer(DPNMSG_DESTROY_PLAYER *Msg);
    BOOL Receive(DPNMSG_RECEIVE *Msg);
};
```

Чтобы начать посылать сообщения `DirectPlay` классу приложения, ваш код переопределяет функции `cServer`, чтобы вызывать соответствующие функции приложения. Чтобы сервер знал, какому экземпляру класса приложения отправлять сообщения, вам необходимо объявить глобальную переменную, указывающую на используемый в данный момент экземпляр класса приложения:

```
cApp *g_Application = NULL;
```

Внутри конструктора наследуемого класса приложения вы затем присваиваете глобальной переменной `g_Application` экземпляр класса приложения:

```
cApp::cApp()
{
    // Прочий код конструктора

    g_Application = this; // Установка указателя экземпляра приложения
}
```

Теперь вы можете закодировать компонент сетевого сервера для отправки входящих сообщений объекту приложения, определенному в глобальном указателе `g_Application`:

```
BOOL cServer::CreatePlayer(DPNMSG_CREATE_PLAYER *Msg)
{
    // Отправка сообщения экземпляру класса приложения
    // (если он есть)
    if(g_Application != NULL)
        g_Application->CreatePlayer(Msg);

    return TRUE;
}

BOOL cServer::DestroyPlayer(DPNMSG_DESTROY_PLAYER *Msg)
{
    // Отправка сообщения экземпляру класса приложения
    // (если он есть)
    if(g_Application != NULL)
        g_Application->DestroyPlayer(Msg);

    return TRUE;
}

BOOL cServer::Receive(DPNMSG_RECEIVE *Msg)
{
    // Отправка сообщения экземпляру класса приложения
    // (если он есть)
    if(g_Application != NULL)
```

```

        g_Application->Receive(Msg);

    return TRUE;
}

```

Компонент сервера теперь завершен и перенаправляет сетевые сообщения классу приложения. Чтобы преобразовать эти сетевые сообщения в относящиеся к игре, класс приложения должен содержать следующие открытые функции:

```

BOOL cApp::CreatePlayer(DPNMSG_CREATE_PLAYER *Msg)
{
    sCreatePlayerMessage cpm;

    // Инициализация данных сообщения
    cpm.Header.Type = MSG_CREATE_PLAYER;
    cpm.Header.Size = sizeof(sCreatePlayerMessage);
    cpm.Header.PlayerID = Msg->dpnidPlayer;

    QueueMessage(&cpm); // Помещаем в очередь сообщений

    return TRUE;
}

BOOL cApp::DestroyPlayer(DPNMSG_DESTROY_PLAYER *Msg)
{
    sDestroyPlayerMessage dpm;

    // Инициализация данных сообщения
    dpm.Header.Type = MSG_DESTROY_PLAYER;
    dpm.Header.Size = sizeof(sDestroyPlayerMessage);
    dpm.Header.PlayerID = Msg->dpnidPlayer;

    QueueMessage(&dpm); // Помещаем в очередь сообщений

    return TRUE;
}

BOOL cApp::Receive(DPNMSG_RECEIVE *Msg)
{
    sMessageHeader *mh = (sMessageHeader*)Msg->pReceiveData;

    // Проверяем, что у сообщения допустимый тип,
    // и помещаем его в очередь
    switch(mh->Type) {
        case MSG_PLAYER_INFO:
        case MSG_STATE_CHANGE:
            // Добавляем сообщение к очереди
            QueueMessage((void*)Msg->pReceiveData);
            break;
    }

    return TRUE;
}

```

Как видите, в каждой из этих трех функций я конструирую относящееся к игре сообщение, используя данные из предоставленного DirectPlay сообщения. Когда игрок пытается подключиться к серверу, создается сообщение *подключения игрока*, хранящее идентификационный номер



DirectPlay подключающегося игрока (вместе с типом сообщения и его размером). Затем это сообщение о создании игрока помещается в очередь.

Когда игроки отключаются от игры, конструируется и помещается в очередь сообщение *отключения игрока*. И, наконец, когда от клиента получены данные (отличные от системных сообщений), функция **cApp::Receive** проверяет указан ли допустимый тип сообщения и, если да, сообщение помещается в очередь.

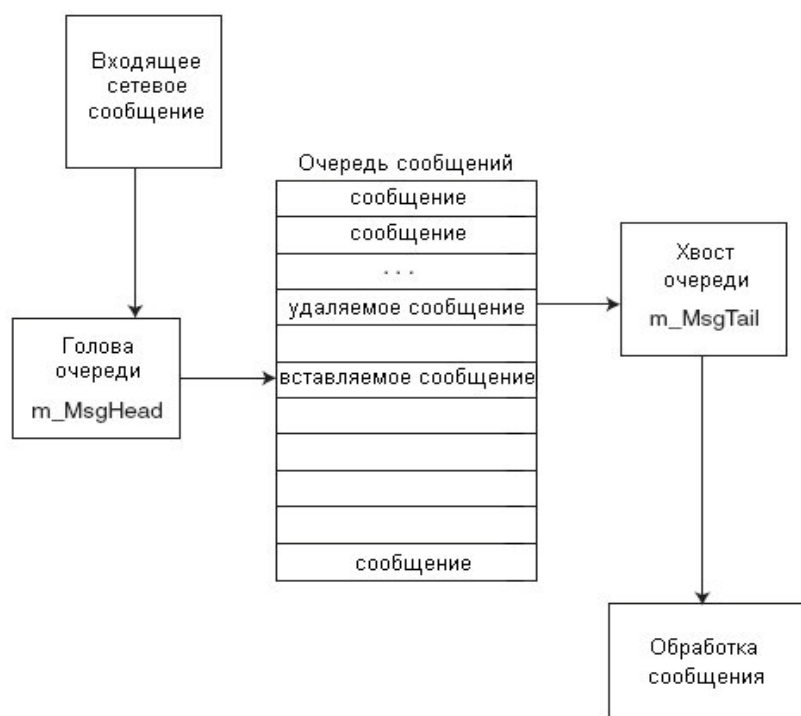
Я продолжаю упоминать очередь сообщений, и показанные выше функции добавляют сообщения в эту очередь. Сейчас вы узнаете чем является эта очередь и как она работает.

## Очередь сообщений

Сервер никогда не имеет дела непосредственно с поступившими сообщениями; вместо этого сервер извлекает сообщения из очереди. Если сообщение должно быть обработано, его необходимо вставить в очередь. Использование очереди гарантирует, что сервер никогда не увязнет, обрабатывая поступающие по сети данные.

Очередь — это просто массив структур **sMessage**, создаваемый при инициализации класса приложения. Для сервера я установил предел очереди равным 1024 сообщениям, но вы можете поменять этот размер просто откорректировав макроопределение **NUM\_MESSAGE** в исходном коде.

Для отслеживания добавляемых и удаляемых сообщений в очереди используются две переменные — **m\_MsgHead** и **m\_MsgTail**. На рис. 15.12 показано, как очередь использует эти две переменные, чтобы отслеживать какие сообщения вставляются или извлекаются.



**Рис. 15.12.** Переменная **m\_MsgHead** отмечает следующую позицию в очереди сообщений для вставки сообщения. Переменная **m\_MsgTail** — это местоположение из которого будет извлекаться сообщение

Каждый раз, когда в очередь необходимо добавить сообщение, вызывается специальная функция. Это функция **cApp::QueueMessage**, и она получает единственный аргумент — структуру **sMessage** для добавления в очередь.

Помните функции входящих сообщений из **cApp** (описанные в разделе «От сообщений DirectPlay к игровым сообщениям»)? Эти функции строят структуры сообщений и добавляют сообщения в очередь через **QueueMessage**. Посмотрите на код **QueueMessage**, чтобы увидеть, что при этом происходит:

```

BOOL cApp::QueueMessage(void *Msg)
{
    sMessageHeader *mh = (sMessageHeader*)Msg;

    // Проверка ошибок - проверяем наличие массива сообщений
    if(m_Messages == NULL)
        return FALSE;

    // Возврат, если в очереди не осталось места
    if(((m_MsgHead + 1) % NUM_MESSAGES) == m_MsgTail)
        return FALSE;

    // Записываем сообщение в очередь
    if(mh->Size <= sizeof(sMessage)) {

        // Начало критической секции
        EnterCriticalSection(&m_MessageCS);

        memcpy(&m_Messages[m_MsgHead], Msg, mh->Size);

        // Переходим к следующему пустому сообщению
        // (если мы в конце очереди, перекидываемся на начало)
        m_MsgHead++;
        if(m_MsgHead >= NUM_MESSAGES)
            m_MsgHead = 0;

        // Покидаем критическую секцию
        LeaveCriticalSection(&m_MessageCS);
    }

    return TRUE;
}

```

Как видите, **QueueMessage** просто копирует предоставленную структуру **sMessage** в следующий доступный элемент массива сообщений (на который указывает **m\_MsgHead**). Вы пока еще не встречались с двумя вещами — функциями **EnterCriticalSection** и **LeaveCriticalSection**.

Windows использует эти две функции чтобы ограничить доступ приложений к памяти (используя функцию **EnterCriticalSection**), позволяя выполнять модификацию памяти только одному процессу. Завершив работу с памятью вы должны сообщить об этом Windows, вызвав **LeaveCriticalSection**.

Хотя сперва это может показаться вам бессмысленным, думайте об этом следующим образом — сетевой компонент (процесс) работает в фоновом режиме одновременно с приложением (другой процесс). Если сетевой компонент будет добавлять сообщения в массив в то же самое время, когда приложение пытается удалить или модифицировать сообщение, данные программы могут оказаться поврежденными. Критические секции гарантируют, что только один процесс на короткое время получит единоличный доступ к данным.

### **Обработка игровых сообщений**

Теперь, когда игровые сообщения проделали свой путь до очереди сообщений, очередным этапом будет извлечение сообщений в каждом кадре и их обработка. Чтобы все работало быстро, за один раз могут обрабатываться только 64 сообщения (это значение задается макроопределением **MESSAGE\_PER\_FRAME** в исходном коде сервера).

Обработка сообщений происходит в функции **cApp::ProcessQueuedMessages**:

```
void cApp::ProcessQueuedMessages()
{
    sMessage *Msg;
    long Count = 0;

    // Извлекаем сообщения для обработки
    while(Count != MESSAGES_PER_FRAME && m_MsgHead != m_MsgTail) {

        // Получаем указатель на "хвостовое" сообщение
        EnterCriticalSection(&m_MessageCS);
        Msg = &m_Messages[m_MsgTail];
        LeaveCriticalSection(&m_MessageCS);

        // Обрабатываем одно сообщение в зависимости от его типа
        switch(Msg->Header.Type) {
            case MSG_PLAYER_INFO: // Запрос информации игрока
                PlayerInfo(Msg, Msg->Header.PlayerID);
                break;

            case MSG_CREATE_PLAYER: // Добавление игрока
                AddPlayer(Msg);
                break;

            case MSG_DESTROY_PLAYER: // Удаление игрока
                RemovePlayer(Msg);
                break;

            case MSG_STATE_CHANGE: // Изменение состояния игрока
                PlayerStateChange(Msg);
                break;
        }

        Count++; // Увеличиваем счетчик обработанных сообщений

        // Переходим к следующему сообщению в списке
        EnterCriticalSection(&m_MessageCS);
        m_MsgTail = (m_MsgTail + 1) % NUM_MESSAGES;
```

```

        LeaveCriticalSection(&m_MessageCS);
    }
}

```

Когда функция **ProcessQueuedMessages** проходит в цикле по очередным 64 сообщениям, она обращается к набору отдельных функций для обработки различных игровых сообщений. Эти функции обработки сообщений описываются в последующих разделах.

### cApp::AddPlayer

Давайте решим — ваша игра будет крутой, и скоро появится множество игроков, присоединяющихся к ней из разных мест. Когда игрок присоединяется к игре (или, по крайней мере, пытается присоединиться), сообщение от игрока добавляется в очередь, и когда это сообщение обрабатывается, будет вызвана функция **AddPlayer**, чтобы найти свободное место для игрока. Если свободного места нет, игрок отключается.

```

BOOL cApp::AddPlayer(sMessage *Msg)
{
    long i;
    DWORD Size = 0;
    DPN_PLAYER_INFO *dpi = NULL;
    HRESULT hr;
    DPNID PlayerID;

    // Контроль ошибок
    if(m_Players == NULL)
        return FALSE;

    PlayerID = Msg->Header.PlayerID;

    // Получаем информацию игрока
    hr = m_Server.GetServerCOM()->GetClientInfo(PlayerID, dpi,
                                                &Size, 0);

    // Возвращаемся при ошибке или если пытаемся добавить сервер
    if(FAILED(hr) && hr != DPNERR_BUFFERTOOSMALL)
        return FALSE;

    // Выделяем память для буфера данных игрока и пытаемся снова
    if((dpi = (DPN_PLAYER_INFO*)new BYTE[Size]) == NULL)
        return FALSE;

    ZeroMemory(dpi, Size);
    dpi->dwSize = sizeof(DPN_PLAYER_INFO);
    if(FAILED(m_Server.GetServerCOM()->GetClientInfo(
                                                PlayerID, dpi, &Size, 0))) {
        delete [] dpi;
        return FALSE;
    }
}

```

К этому моменту сервер запросил у DirectPlay информацию о клиенте (как она была установлена клиентом), которая включает имя клиента. Теперь сервер сканирует массив структур **sPlayer**, ища такую, где флаг **Connected** равен **FALSE** (и также проверяя, что игрок уже не подключен) и, значит, данный слот свободен для присоединения игрока.

```
// Проверяем, что игрока нет в списке
for(i = 0; i < MAX_PLAYERS; i++) {
    if(m_Players[i].dpnidPlayer == PlayerID &&
        m_Players[i].Connected == TRUE) {
        delete [] dpi;
        m_Server.DisconnectPlayer(PlayerID);
        return FALSE;
    }
}

// Ищем свободный слот для размещения игрока
for(i = 0; i < MAX_PLAYERS; i++) {
    if(m_Players[i].Connected == FALSE) {
        m_Players[i].Connected = TRUE; // Флаг подключения

        // Сохраняем DPNID # DirectPlay и имя игрока
        m_Players[i].dpnidPlayer = PlayerID;
        wcstombs(m_Players[i].Name, dpi->pwszName, 256);

        // Устанавливаем данные игрока
        m_Players[i].XPos = 0.0f;
        m_Players[i].YPos = 0.0f;
        m_Players[i].ZPos = 0.0f;
        m_Players[i].Direction = 0.0f;
        m_Players[i].Speed = 512.0f;
        m_Players[i].State = STATE_IDLE;
        m_Players[i].Latency = 0;
    }
}
```

Если в массиве игроков найден свободный слот, сохраняется информация о клиенте и в структуру данных игрока записываются исходные значения. В оставшейся части функции выполняется рассылка всем остальным подключенным игрокам игрового сообщения **MSG\_CREATE\_PLAYER**, сообщающего им о новом игроке.

```
// Рассылаем информацию о добавленном игроке
// всем остальным игрокам
sCreatePlayerMessage cpm;

cpm.Header.Type = MSG_CREATE_PLAYER;
cpm.Header.Size = sizeof(sCreatePlayerMessage);
cpm.Header.PlayerID = PlayerID;
cpm.XPos = m_Players[i].XPos;
cpm.YPos = m_Players[i].YPos;
cpm.ZPos = m_Players[i].ZPos;
cpm.Direction = m_Players[i].Direction;

SendNetworkMessage(&cpm, DPNSEND_NOLOOPBACK, -1);

// Увеличиваем количество игроков
m_NumPlayers++;

ListPlayers(); // Список всех игроков

delete [] dpi; // Освобождаем данные игрока

return TRUE; // Возвращаем флаг успеха
}

delete[] dpi; // Освобождаем данные игрока
```

```

    // Отключаем игрока - соединение запрещено
    m_Server.DisconnectPlayer(PlayerID);

    return FALSE; // Возвращаем флаг неудачи
}

```

## cApp::RemovePlayer

Реализовав присоединение игроков к игре, дайте им возможность и выйти из нее, и это цель функции **RemovePlayer**. В функции **RemovePlayer** сервер сканирует список подключенных игроков, ища совпадение идентификационного номера DirectPlay (с отключаемым игроком) и удаляет этого игрока из списка. После завершения сканирования и удаления соответствующего игрока из списка все клиенты уведомляются об отключении игрока и сервер заново строит список существующих игроков.

```

BOOL cApp::RemovePlayer(sMessage *Msg)
{
    long i;

    // Контроль ошибок
    if(m_Players == NULL)
        return FALSE;

    // Поиск игрока в списке
    for(i = 0; i < MAX_PLAYERS; i++) {
        if(m_Players[i].dpnidPlayer == Msg->Header.PlayerID &&
            m_Players[i].Connected == TRUE) {

            // Отключение игрока
            m_Players[i].Connected = FALSE;

            // Отправка сообщения об удалении игрока
            // всем остальным игрокам
            sDestroyPlayerMessage dpm;

            dpm.Header.Type = MSG_DESTROY_PLAYER;
            dpm.Header.Size = sizeof(sDestroyPlayerMessage);
            dpm.Header.PlayerID = Msg->Header.PlayerID;

            SendNetworkMessage(&dpm, DPNSEND_NOLOOPBACK, -1);

            // Уменьшение количества игроков
            m_NumPlayers--;

            // Список всех игроков
            ListPlayers();

            return TRUE;
        }
    }

    return FALSE; // Возврат ошибки
}

```

### **cApp::PlayerInfo**

К сожалению, в сетевых играх игровые сообщения иногда теряются в пути. Что если одно из этих потерянных сообщений намеревалось сообщить клиентскому приложению о присоединении к игре нового игрока? Более того, что случится, если клиент начнет получать сообщения, относящиеся к игроку, о существовании которого он не знает (из-за потерянного сообщения)?

В случае, если клиент не знает об игроке и получает относящиеся к нему сообщения, клиенту необходимо запросить данные соответствующего игрока от сервера, чтобы продолжить работу. Сервер, в свою очередь, отправляет запрошенную информацию игрока клиенту, используя функцию **PlayerInfo**:

```
BOOL cApp::PlayerInfo(sMessage *Msg, DPNID To)
{
    sRequestPlayerInfoMessage *rpim;
    sCreatePlayerMessage      cpm;
    long i;

    // Проверка ошибок
    if(m_Players == NULL)
        return FALSE;

    // Получаем указатель на запрашиваемую информацию
    rpim = (sRequestPlayerInfoMessage*)Msg;

    for(i = 0; i < MAX_PLAYERS; i++) {

        // Отправляем только если нашли в списке
        if(m_Players[i].dpnidPlayer == rpim->PlayerID &&
           m_Players[i].Connected == TRUE) {

            // Отправляем информацию игрока тому, кто ее запросил
            cpm.Header.Type = MSG_PLAYER_INFO;
            cpm.Header.Size = sizeof(sCreatePlayerMessage);
            cpm.Header.PlayerID = rpim->PlayerID;
            cpm.XPos = m_Players[i].XPos;
            cpm.YPos = m_Players[i].YPos;
            cpm.ZPos = m_Players[i].ZPos;
            cpm.Direction = m_Players[i].Direction;

            SendNetworkMessage(&cpm, DPNSEND_NOLOOPBACK, To);

            break;
        }
    }

    return TRUE;
}
```

### **cApp::PlayerStateChange**

Главной функцией обработки сообщений на стороне сервера является **PlayerStateChange**, получающая входящие действия от клиентов и обновляющая внутренние данные игроков.

```

BOOL cApp::PlayerStateChange(sMessage *Msg)
{
    sStateChangeMessage *scm, uscm;
    long i, PlayerNum;
    BOOL AllowChange;
    float XDiff, ZDiff, Dist, Angle;

    // Проверка ошибок
    if(m_Players == NULL)
        return FALSE;

    // Получаем указатель на сообщение об изменении состояния
    scm = (sStateChangeMessage*)Msg;

    // Получаем номер игрока в списке
    PlayerNum = -1;

    for(i = 0; i < MAX_PLAYERS; i++) {
        if(m_Players[i].dpnidPlayer == Msg->Header.PlayerID &&
           m_Players[i].Connected == TRUE) {
            PlayerNum = i;
            break;
        }
    }

    if(PlayerNum == -1)
        return FALSE;

```

К этому моменту сервер выполнил поиск игрока, приславшего сообщение об изменении состояния. Если сообщение пришло от игрока, который не подключен, оно игнорируется. Далее вступает в действие игровая логика.

Игрокам разрешено ходить, стоять и размахивать своим оружием. Те игроки, кто уже начал размахивать оружием или подвергся нападению не могут изменять свое состояние (пока их текущее состояние не будет очищено).

```

AllowChange = TRUE; // Флаг разрешения изменения состояния

// Отказываемся обновлять игрока, размахивающего мечом
if(m_Players[PlayerNum].State == STATE_SWING)
    AllowChange = FALSE;

// Отказываемся обновлять атакованного игрока
if(m_Players[PlayerNum].State == STATE_HURT)
    AllowChange = FALSE;

// Только если разрешено обновление состояния
if(AllowChange == TRUE) {
    // Обновляем выбранного игрока
    m_Players[PlayerNum].Time = timeGetTime();
    m_Players[PlayerNum].State = scm->State;
    m_Players[PlayerNum].Direction = scm->Direction;

    // Подстраиваем время действия, согласно запаздыванию
    m_Players[PlayerNum].Time -= m_Players[PlayerNum].Latency;

    // Отправляем данные игрока всем клиентам
    uscm.Header.Type = MSG_STATE_CHANGE;

```



```
uscm.Header.Size = sizeof(sStateChangeMessage);
uscm.Header.PlayerID = scm->Header.PlayerID;
uscm.State = m_Players[PlayerNum].State;
uscm.XPos = m_Players[PlayerNum].XPos;
uscm.YPos = m_Players[PlayerNum].YPos;
uscm.ZPos = m_Players[PlayerNum].ZPos;
uscm.Direction = m_Players[PlayerNum].Direction;
uscm.Speed = m_Players[PlayerNum].Speed;
```

```
SendNetworkMessage(&uscm, DPNSSEND_NOLOOPBACK);
```

Теперь состояние игрока обновлено (если это разрешено) и отправлено всем остальным подключенным игрокам. Затем, если игрок взмахивает своим оружием, сканируются все игроки, чтобы увидеть, не задел ли атакующий кого-нибудь. Если да, состояние жертвы меняется на **HURT**.

Также заметьте, что я смещаю значение переменной времени состояния (**sPlayer::Time**) на значение запаздывания игрока (**sPlayer::Latency**). Эта подстройка для задержек передачи по сети улучшает синхронизацию. Если вы удалите смещение на величину запаздывания, то увидите эффект скачков, когда игроки будут перемещаться по уровню.

```
// Если игрок взмахнул мечом, определяем жертву
if(scm->State == STATE_SWING) {

    // Проверяем всех игроков
    for(i = 0; i < MAX_PLAYERS; i++) {

        // Проверяем только других подключенных игроков
        if(i != PlayerNum && m_Players[i].Connected == TRUE) {
            // Получаем расстояние до игрока
            XDiff = (float)fabs(m_Players[PlayerNum].XPos -
                               m_Players[i].XPos);
            ZDiff = (float)fabs(m_Players[PlayerNum].ZPos -
                               m_Players[i].ZPos);
            Dist = XDiff * XDiff + ZDiff * ZDiff;

            // Продолжаем, если расстояние приемлемо
            if(Dist < 10000.0f) {
                // Получаем угол между игроками
                Angle = -(float)atan2((m_Players[i].ZPos -
                                       m_Players[PlayerNum].ZPos),
                                       (m_Players[i].XPos -
                                       m_Players[PlayerNum].XPos)) +
                    1.570796f;

                // Подстраиваем направление атакующего
                Angle -= m_Players[PlayerNum].Direction;
                Angle += 0.785f; // Подстройка для FOV

                // Ограничиваем значения углов
                if(Angle < 0.0f)
                    Angle += 6.28f;
                if(Angle >= 6.28f)
                    Angle -= 6.28f;

                // Игрок поражен, если он перед атакующим (90 FOV)
                if(Angle >= 0.0f && Angle <= 1.57f) {
```

Заметьте, что размахивающий мечом игрок может поразить только игрока, находящегося перед ним. Чтобы проверить, был ли другой игрок поражен во время атаки, вы сперва вычисляете расстояние, и, если персонажи находятся достаточно близко друг к другу, проверяете угол между игроками. Если атакуемый игрок находится в пределах 90-градусного поля зрения перед атакующим (как показано на рис. 15.13), его считают пораженным, и с этого момента состояние жертвы меняется на **HURT**.



*Рис. 15.13. Чтобы игрок мог поразить другого игрока атакующий должен находиться достаточно близко и быть обращен лицом к предполагаемой жертве. Сервер проверяет, находится ли жертва в 90-градусном поле зрения атакующего*

```
// Установить жертве состояние ранения
// (если это еще не сделано)
if(m_Players[i].State != STATE_HURT) {
    m_Players[i].State = STATE_HURT;
    m_Players[i].Time = timeGetTime();

    // Отправляем сетевое сообщение
    uscm.Header.Type = MSG_STATE_CHANGE;
    uscm.Header.Size =
        sizeof(sStateChangeMessage);
    uscm.Header.PlayerID =
        m_Players[i].dpnidPlayer;
    uscm.State = m_Players[i].State;
    uscm.XPos = m_Players[i].XPos;
    uscm.YPos = m_Players[i].YPos;
    uscm.ZPos = m_Players[i].ZPos;
    uscm.Direction = m_Players[i].Direction;
    uscm.Speed = m_Players[i].Speed;

    SendNetworkMessage(&uscm,
        DPNSSEND_NOLOOPBACK);
}
}
}
}
}
}
}
}
return TRUE;
}
```

Вот и все необходимое для того, чтобы иметь дело с игровыми сообщениями и изменением состояния игроков. Хотя функция **PlayerStateChange** ответственна за разбор помещенных в очередь игровых сообщений, действительное перемещение игроков и очистка их состояния размахивания оружием и ранения происходит в другой функции, которую вы увидите в следующем разделе.

## Обновление игроков

Чтобы синхронизироваться с клиентами серверу необходимо внутри поддерживать запущенную упрощенную версию игры. Эта версия игры не включает в себя графику, звук и любые другие мультимедийные возможности; ей необходимо только отслеживать действия игроков.

Сервер выполняет это отслеживание действий, обновляя текущие действия игроков каждые 33 мс (точно так же, как это делает клиентское приложение). Эти действия включают ходьбу и ожидание очистки других специальных действий (таких, как размахивание мечом и ранение).

За обновление всех игроков отвечает функция **cApp::UpdatePlayers**:

```
void cApp::UpdatePlayers()
{
    long i;
    float XMove, ZMove, Speed;
    sStateChangeMessage scm;
    long Elapsed;

    // Цикл перебора всех игроков
    for(i = 0; i < MAX_PLAYERS; i++) {
        // Обновляем только подключенных игроков
        if(m_Players[i].Connected == TRUE) {

            // Получаем время, прошедшее от обновления состояния
            // до текущего момента
            Elapsed = timeGetTime() - m_Players[i].Time;
```

Сервер сканирует список игроков, определяя какие игроки подключены и вычисляя для всех подключенных игроков время, прошедшее с последнего обновления сервера. Затем, если состояние игрока установлено в **STATE\_MOVE**, вычисленный период времени используется для перемещения игрока:

```
        // Обработка состояния движения игрока
        if(m_Players[i].State == STATE_MOVE) {

            // Вычисляем перемещение на основе прошедшего времени
            Speed = (float)Elapsed / 1000.0f * m_Players[i].Speed;
            XMove = (float)sin(m_Players[i].Direction) * Speed;
            ZMove = (float)cos(m_Players[i].Direction) * Speed;

            // Проверка столкновений при движении -
            // нельзя проходить сквозь блокирующие путь препятствия
            if(CheckIntersect(&m_LevelMesh,
                            m_Players[i].XPos,
```



Удивительно, но это все об **cApp::UpdatePlayers**! Помните, что функция **UpdatePlayers** вызывается каждые 33 мс, так что сохраняйте ее быстрой и помните, что она является критической точкой. Как только все игроки обновлены, вам надо уведомить других игроков.

## Обновление сетевых клиентов

Во всех предыдущих разделах этой главы я упоминал серверные обновления, рассылаемые клиентам для синхронизации игрового процесса. В этом и заключается назначение функции **cApp::UpdateNetwork**. Быстрая и прямолинейная функция **UpdateNetwork** рассылает каждые 100 мс текущее состояние всем подключенным клиентам.

```
void cApp::UpdateNetwork()
{
    long i;
    sStateChangeMessage scm;

    // Отправляем обновление всем игрокам
    for(i = 0; i < MAX_PLAYERS; i++) {

        // Рассылаем данные только о подключенных игроках
        if(m_Players[i].Connected == TRUE) {
            scm.Header.Type = MSG_STATE_CHANGE;
            scm.Header.Size = sizeof(sStateChangeMessage);
            scm.Header.PlayerID = m_Players[i].dpnidPlayer;
            scm.XPos = m_Players[i].XPos;
            scm.YPos = m_Players[i].YPos;
            scm.ZPos = m_Players[i].ZPos;
            scm.Direction = m_Players[i].Direction;
            scm.Speed = m_Players[i].Speed;
            scm.State = m_Players[i].State;
            scm.Latency = m_Players[i].Latency;

            // Отправляем сообщение по сети
            SendNetMessage(&scm, DPNSEND_NOLOOPBACK);
        }
    }
}
```

## Вычисление запаздывания

Сервер периодически вычисляет время, требуемое сообщениям, чтобы быть доставленными от клиента, и использует это запаздывание в расчетах времени при обновлении клиента, что является критически важным для поддержания синхронизации игры. Функция вычисления запаздывания называется **UpdateLatency** и вызывается каждые 10 секунд из главного цикла приложения(**cApp::Frame**).

```
void cApp::UpdateLatency()
{
    long i;
    DPN_CONNECTION_INFO dpci;
    HRESULT hr;
```

```

// Перебираем всех игроков
for(i = 0; i < MAX_PLAYERS; i++) {

    // Обрабатываем только подключенных игроков
    if(m_Players[i].Connected == TRUE) {

        // Запрашиваем параметры подключения игрока
        hr = m_Server.GetServerCOM()->GetConnectionInfo(
            m_Players[i].dpnidPlayer, &dpci, 0);
        if(SUCCEEDED(hr)) {
            m_Players[i].Latency = dpci.dwRoundTripLatencyMS / 2;

            // Ограничиваем запаздывание 1 секундой
            if(m_Players[i].Latency > 1000)
                m_Players[i].Latency = 1000;
        } else {
            m_Players[i].Latency = 0;
        }
    }
}
}

```

Для вычисления запаздывания сервер запрашивает у DirectPlay статистику соединения через функцию **IDirectPlay8Server::GetConnectInfo**. Эта функция получает в аргументе структуру (**DPN\_CONNECTION\_INFO**), в которой есть переменная, представляющая время запаздывания в оба конца в миллисекундах. Сервер делит это значение запаздывания пополам и сохраняет в структуре данных каждого игрока.

## Самое трудное позади!

Вы закончили разбираться во внутренностях сервера. Осталось только завернуть все это в полнофункциональное приложение. Помимо кода сервера осталось сделать совсем немного. Чтобы увидеть как настраивается приложение, посмотрите код сервера на CD-ROM. Теперь пришло время сосредоточить внимание на другой стороне сетевой игры — клиенте!

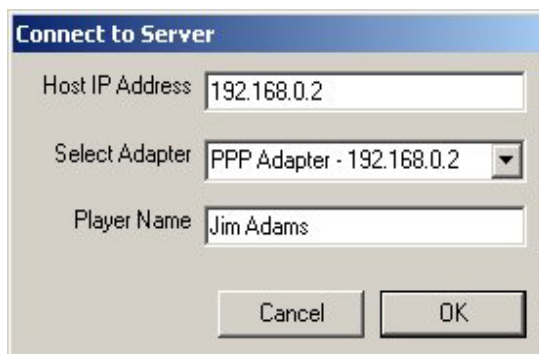
## Работа над игровым клиентом

Клиентское приложение (далее мы будем называть его просто *клиент*) является средством коммуникации между игровым сервером и игроком. Клиент получает входные данные от пользователя и переправляет их серверу. В промежутках между получениями обновлений от сервера клиент самостоятельно выполняет обновление, основываясь на той скромной информации, которая у него есть — перемещении игрока, передвижении других игроков, действиях NPC и т.д.

Чтобы волшебство заработало клиент использует графику, звук и обработку ввода. Если отказаться от графики и звука, останется весьма примитивное приложение. Такой «тупой» каркас клиента может выглядеть не имеющим ценности, но поверьте мне, он будет замечательно работать в ваших игровых проектах.

Для использования клиентского приложения (оно находится на прилагаемом к книге CD-ROM в каталоге \BookCode\Chap15\Client), выполните следующие действия:

1. Найдите и запустите приложение **Client**. Появится диалоговое окно **Connect to Server** (рис. 15.14).



*Рис. 15.14. Вам надо выбрать адаптер и ввести имя игрока, а также необходимо знать IP-адрес сервера, чтобы подключиться и начать игру*

2. В диалоговом окне **Connect to Server** введите IP-адрес главного узла, выберите адаптер и введите имя вашего игрока.
3. Щелкните **ОК**, чтобы начать игру и подключиться к серверу. В некоторых отношениях клиент работает почти идентично серверу и, в частности, это относится к работе с игроками.

## Обработка данных игрока

Клиент, также как и сервер, использует структуру **sPlayer**, содержащую информацию о каждом подключенном к игре игроке. Однако на этот раз информация нужна для отслеживания трехмерных объектов, чтобы рисовать игрока (а также его оружие) и выполнять анимацию в ходе игры. Помимо этого вы увидите много общего между структурами **sPlayer**, используемыми на стороне клиента и сервера. Взгляните на объявление клиентской структуры **sPlayer** (вместе с вспомогательными макроопределениями):

```
// Состояния игрока
#define STATE_IDLE 1
#define STATE_MOVE 2
#define STATE_SWING 3
#define STATE_HURT 4

// Анимации
#define ANIM_IDLE 1
#define ANIM_WALK 2
#define ANIM_SWING 3
#define ANIM_HURT 4

typedef struct sPlayer {
    BOOL Connected; // TRUE если игрок активен
    DPNID dpnidPlayer; // ID игрока в DirectPlay

    long State; // Последнее известное состояние (STATE_*)
```

```

long Time;           // Время последнего обновления состояния
long Latency;        // Половина полного цикла запаздывания в мс

float XPos, YPos, ZPos; // Координаты игрока
float Direction;       // Угол направления взгляда
float Speed;           // Скорость перемещения

cObject Body;         // 3-D объект персонажа
cObject Weapon;       // 3-D объект оружия

long LastAnim;        // Последняя установленная анимация

// Конструктор и деструктор
sPlayer()
{
    Connected = FALSE;
    dpnidPlayer = 0;
    LastAnim = 0;
    Time = 0;
}

~sPlayer()
{
    Body.Free();
    Weapon.Free();
}
} sPlayer;

```

И снова для хранения информации об игроках выделяется массив структур **sPlayer**. Каждый игрок использует отдельные объекты графического ядра для представления сеток тела игрока и его оружия. Локальный игрок использует первый элемент массива данных игроков (определенного в классе приложения как **m\_Players**), а присоединяющиеся игроки помещаются в первый найденный свободный слот.

В ходе инициализации класса приложения клиента, сетки для всех персонажей и вооружений загружаются и присваиваются членам структур данных каждого игрока. Это первый шанс усовершенствовать вашу сетевую игру; загружая различные сетки вы можете сделать так, чтобы каждый игрок выглядел по-своему. Например, один персонаж может быть воином, другой — волшебником, и т.д.

Также загружается список анимаций. Эти анимации представляют различные состояния игроков: анимация ходьбы, стояние (ожидание), размахивание оружием и, наконец, анимация ранения. Анимации устанавливаются функцией **UpdatePlayers**, которую вы увидите чуть позже в разделе «Обновление локального игрока».

Яркой деталью структуры **sPlayer** является идентификационный номер **DirectPlay**. Клиентам разрешено хранить свои собственные локальные идентификаторы игрока для тех случаев, когда им надо отправить сообщение о смене состояния серверу; идентификационный номер игрока является частью сообщения смены состояния.



Для получения своего собственного идентификационного номера игрока клиент должен обработать сообщение **DPN\_MSGID\_CONNECTION\_COMPLETE**, отправляемое когда клиент успешно подключился к серверу. Вы увидите как извлечь идентификационный номер локального игрока в следующем разделе, «Сетевой компонент». Получив идентификационный номер игрока в клиенте вы можете использовать его, чтобы определить к каким подключенным игрокам относятся входящие сообщения об обновлении.

Получив от сервера игровое сообщение, клиентское приложение перебирает список подключенных игроков. Когда идентификационный номер игрока в локальном списке совпадает с полученным от сервера, клиент точно определяет данные какого игрока следует обновить.

Для перебора списка игроков клиент использует функцию с именем **GetPlayerNum**, возвращающую индекс соответствующего игрока в массиве (или -1, если совпадений не обнаружено):

```
long cApp::GetPlayerNum(DPNID dpnidPlayer)
{
    long i;

    // Контроль ошибок
    if(m_Players == NULL)
        return -1;

    // Поиск совпадения в списке
    for(i = 0; i < MAX_PLAYERS; i++) {
        if(m_Players[i].dpnidPlayer == dpnidPlayer &&
           m_Players[i].Connected == TRUE)
            return i;
    }

    return -1; // В списке ничего не найдено
}
```

Теперь клиент всегда может воспользоваться функцией **GetPlayerNum**, чтобы определить, какого игрока обновлять. Если игрок не найден в списке, но точно известно, что он существует, клиент должен отправить сообщение **MSG\_PLAYER\_INFO**, запрашивающее информацию об игроке у сервера. В ответ сервер вернет пославшему запрос клиенту сообщение создания игрока.

Я немного забежал вперед, так что давайте замедляться. Также как и сервер, для обработки сетевых подключений клиент использует сетевое ядро. Сейчас мы бросим взгляд на клиентский сетевой компонент, используемый в клиентских приложениях.

## Сетевой компонент

Чтобы использовать клиентский компонент вы наследуете от него свой класс и в этом классе-наследнике переопределяете необходимые функции. Таких функций немного и они нужны только чтобы уведомлять об установлении подключения к серверу и получать входящие игровые сообщения.

Использование клиентского сетевого компонента начнем с наследования нашего собственного класса от **cNetworkClient**:

```
class cClient : public cNetworkClient
{
    private:
        BOOL ConnectComplete(DPNMSG_CONNECT_COMPLETE *Msg);
        BOOL Receive(DPNMSG_RECEIVE *Msg);
};
```

Поскольку сообщения получает объект сетевого клиента (через функцию **cClient::Receive**), они должны быть переданы объекту вашего приложения. Следовательно цель функций **cClient::Receive** и **cApp::Receive** — служить каналом от сетевого объекта до объекта приложения для сетевых сообщений:

```
BOOL cClient::Receive(DPNMSG_RECEIVE *Msg)
{
    // Отправка сообщения экземпляру класса приложения
    // (если он есть)
    if(g_Application != NULL)
        g_Application->Receive(Msg);

    return TRUE;
}

BOOL cApp::Receive(DPNMSG_RECEIVE *Msg)
{
    sMessage *MsgPtr;

    // Получаем указатель на полученные данные
    MsgPtr = (sMessage*)Msg->pReceiveData;

    // Обрабатываем пакеты в зависимости от их типа
    switch(MsgPtr->Header.Type) {
        case MSG_PLAYER_INFO: // Добавляем игрока к списку
        case MSG_CREATE_PLAYER:
            CreatePlayer(MsgPtr);
            break;

        case MSG_DESTROY_PLAYER: // Удаляем игрока из списка
            DestroyPlayer(MsgPtr);
            break;

        case MSG_STATE_CHANGE: // Меняем состояние игрока
            ChangeState(MsgPtr);
            break;
    }

    return TRUE;
}
```

Как видите, внутри функции **cClient::Receive** вы передаете сетевое сообщение в функцию **cApp::Receive**. В функции **cApp::Receive** вы обрабатываете четыре различных сообщения: **MSG\_PLAYER\_INFO**, **MSG\_CREATE\_PLAYER**, **MSG\_DESTROY\_PLAYER** и **MSG\_STATE\_CHANGE**.

В отличие от сервера, клиентское приложение не помещает никакие сообщения в очередь; их сразу обрабатывают, используя набор функций. Я возвращусь к сообщениям через мгновение, а сейчас хочу немного отступить назад и исследовать функцию **ConnectComplete**, определенную в сетевом классе.

Функцию **ConnectComplete** вызывают всякий раз, когда клиент успешно завершит установление соединения с сервером. Внутри функции **ConnectComplete** вы сохраняете флаг подключения в глобальной переменной **g\_Connected**, и идентификационный номер игрока, использующего данное клиентское приложение:

```
cApp *g_Application; // Глобальный указатель на объект приложения

BOOL cClient::ConnectComplete(DPNMSG_CONNECT_COMPLETE *Msg)
{
    // Сохраняем флаг подключения
    if(Msg->hResultCode == S_OK)
        g_Connected = TRUE;
    else
        g_Connected = FALSE;

    // Получаем идентификатор игрока из кода завершения подключения
    if(g_Application != NULL)
        g_Application->SetLocalPlayerID(Msg->dpnidLocal);

    return TRUE;
}
```

Здесь есть кое-что интересное — вызов функции **SetLocalPlayerID** вашего класса приложения. Спрашиваете, что это за функция? Помните, как вы использовали функцию **cApp::Receive** для проталкивания сетевых сообщений в объект вашего приложения? То же можно сказать и о функции **cApp::SetLocalPlayerID** — она передает идентификационный номер локального игрока в класс приложения:

```
BOOL cApp::SetLocalPlayerID(DPNID dpnid)
{
    if(m_Players == NULL)
        return FALSE;

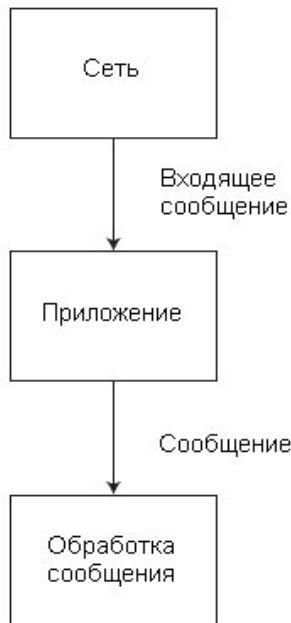
    EnterCriticalSection(&m_UpdateCS);
    m_Players[0].dpnidPlayer = dpnid;
    LeaveCriticalSection(&m_UpdateCS);

    return TRUE;
}
```

Функция **SetLocalPlayerID** вызывается непосредственно из сетевого объекта — она обеспечивает установку идентификационного номера локального игрока, чтобы ваша игра имела доступ к нему когда будет обрабатывать сообщения об изменении состояния. Как только у вас появится этот идентификационный номер игрока, станет возможной обработка других сетевых сообщений, о чем вы сейчас прочтете.

## Обработка сообщений

Клиентское приложение использует те же самые структуры сообщений что и сервер, но, как я ранее упоминал, клиент не нуждается в очереди сообщений. На рис. 15.15 показан немедленный разбор входящих сообщений клиентом.



*Рис. 15.15. Клиент получает сообщения от клиентского сетевого компонента точно так же, как и сервер. Однако клиент сразу преобразовывает входящие сообщения в игровые и обрабатывает их*

Вы уже видели, как функция **Receive** обрабатывает входящие сообщения, и пришло время исследовать каждую из функций обработчиков сообщений.

### **cApp::CreatePlayer**

Когда клиент присоединяется к игре сервер информирует других подключенных клиентов о вновь прибывшем. Цель приведенной ниже функции **CreatePlayer** заключается в отыскании места для структуры **sPlayer** и сохранении данных игрока (помните, что перебирая элементы массива **m\_Players** вы пропускаете элемент с индексом 0, поскольку он зарезервирован для данных локального игрока):

```

void cApp::CreatePlayer(sMessage *Msg)
{
    sCreatePlayerMessage *cpm;
    long PlayerNum, i;

    // Контроль ошибок
    if(m_Players == NULL || !m_Players[0].dpnidPlayer)
        return;

    // Получаем указатель на данные сообщения
    cpm = (sCreatePlayerMessage*)Msg;

    // Локального игрока добавлять в список не надо
    if(cpm->Header.PlayerID == m_Players[0].dpnidPlayer)
        return;
  
```

```
// Проверяем, что игрока нет в списке
// и одновременно ищем свободный слот
PlayerNum = -1;

// Перебираем список, пропустив локального игрока (слот 0)
for(i = 1; i < MAX_PLAYERS; i++) {
    if(m_Players[i].Connected == TRUE) {
        if(m_Players[i].dpnidPlayer == cpm->Header.PlayerID)
            return;
    } else
        PlayerNum = i;
}

// Ошибка - нет свободных слотов
if(PlayerNum == -1)
    return;

// Начало критической секции
EnterCriticalSection(&m_UpdateCS);

// Добавляем данные игрока
m_Players[PlayerNum].Connected = TRUE;
m_Players[PlayerNum].dpnidPlayer = cpm->Header.PlayerID;
m_Players[PlayerNum].XPos = cpm->XPos;
m_Players[PlayerNum].YPos = cpm->YPos;
m_Players[PlayerNum].ZPos = cpm->ZPos;
m_Players[PlayerNum].Direction = cpm->Direction;
m_Players[PlayerNum].Speed = 0.0f;
m_Players[PlayerNum].State = STATE_IDLE;
m_NumPlayers++;

// Покидаем критическую секцию
LeaveCriticalSection(&m_UpdateCS);
}
```

### ***cApp::DestroyPlayer***

Сервер уведомляет клиентов когда игрок завершает сессию. Клиент, в свою очередь, отмечает, что игрок начал отключение и пропускает обновление данных игрока в цикле обновления. Показанный ниже код определяет, какой клиент отключился, и предпринимает необходимые шаги:

```
void cApp::DestroyPlayer(sMessage *Msg)
{
    sDestroyPlayerMessage *dpm;
    long PlayerNum;

    // Контроль ошибок
    if(m_Players == NULL || !m_Players[0].dpnidPlayer)
        return;

    // Получаем указатель на данные сообщения
    dpm = (sDestroyPlayerMessage*)Msg;

    // Не удалять из списка локального игрока
    if(dpm->Header.PlayerID == m_Players[0].dpnidPlayer)
        return;

    // Получаем номер игрока в списке
    if((PlayerNum = GetPlayerNum(dpm->Header.PlayerID)) == -1)
        return;
}
```

```

// Начало критической секции
EnterCriticalSection(&m_UpdateCS);

// Помечаем игрока как отключившегося
m_Players[PlayerNum].Connected = FALSE;
m_NumPlayers--;

// Покидаем критическую секцию
LeaveCriticalSection(&m_UpdateCS);
}

```

### **cApp::ChangeState**

Клиент обрабатывает изменения состояния игроков извлекая данные из сообщения и помещая их в структуры данных игроков. Если игрок не найден в списке игроков, клиент запрашивает информацию об игроке через сообщение **MSG\_PLAYER\_INFO** и выходит из функции **ChangeState** без дополнительной суеты.

Это единственная ситуация, когда координаты игрока могут быть непосредственно модифицированы сменой состояния — клиентам не разрешается выполнять непосредственное изменение своих координат (чтобы избежать мошенничества), поэтому сервер должен во время обновлений говорить игрокам где они находятся в мире:

```

void cApp::ChangeState(sMessage *Msg)
{
    sStateChangeMessage *scm;
    sRequestPlayerInfoMessage rpim;
    long PlayerNum;

    // Контроль ошибок
    if(m_Players == NULL || !m_Players[0].dpnidPlayer)
        return;

    // Получаем указатель на данные сообщения
    scm = (sStateChangeMessage*)Msg;

    // Получаем номер игрока в списке
    if((PlayerNum = GetPlayerNum(scm->Header.PlayerID)) == -1) {

        // Неизвестный игрок - запрашиваем информацию
        if(PlayerNum == -1) {
            // Конструируем сообщение
            rpim.Header.Type = MSG_PLAYER_INFO;
            rpim.Header.Size = sizeof(sRequestPlayerInfoMessage);
            rpim.Header.PlayerID = m_Players[0].dpnidPlayer;
            rpim.PlayerID = scm->Header.PlayerID;

            // Отправляем сообщение серверу
            SendNetworkMessage(&rpim, DPNSSEND_NOLOOPBACK);

            return;
        }
    }

    // Начало критической секции
    EnterCriticalSection(&m_UpdateCS);
}

```

```
// Сохраняем информацию о новом состоянии
m_Players[PlayerNum].Time = timeGetTime();
m_Players[PlayerNum].State = scm->State;
m_Players[PlayerNum].XPos = scm->XPos;
m_Players[PlayerNum].YPos = scm->YPos;
m_Players[PlayerNum].ZPos = scm->ZPos;
m_Players[PlayerNum].Direction = scm->Direction;
m_Players[PlayerNum].Speed = scm->Speed;
m_Players[PlayerNum].Latency = scm->Latency;

// Ограничиваем запаздывание одной секундой
if(m_Players[PlayerNum].Latency > 1000)
    m_Players[PlayerNum].Latency = 1000;

// Подстраиваем время в зависимости от запаздывания
m_Players[PlayerNum].Time -= m_Players[PlayerNum].Latency;

// Покидаем критическую секцию
LeaveCriticalSection(&m_UpdateCS);
}
```

Также как у сервера, у клиента есть функция **SendNetworkMessage** для отправки относящихся к игре сетевых сообщений серверу (за дополнительными подробностями работы этой функции обратитесь к исходному коду клиента на прилагаемом к книге CD-ROM).

## Обновление локального игрока

Между обновлениями от сервера клиенту надо обновлять всех игроков, чтобы поддерживать плавность игры. Клиентское приложение производит обновления каждые 33 мс (30 раз в секунду), что соответствует частоте обновлений на сервере. Между этими обновлениями игрока клиенту разрешено собирать входные данные от пользователя, используемые для изменения состояния его персонажа.

Для обновления локального игрока используется функция **cApp::Frame**. Для управления персонажами игроки используют клавиатуру и мышь, поэтому я добавил пару объектов ядра ввода (**m\_Keyboard** и **m\_Mouse**):

```
BOOL cApp::Frame()
{
    static DWORD UpdateCounter = timeGetTime();
    static long MoveAction = 0, LastMove = 0;
    static BOOL CamMoved = FALSE;
    BOOL AllowMovement;
    long Dir;

    float Angles[13] = { 0.0f, 0.0f, 1.57f, 0.785f, 3.14f,
                        0.0f, 2.355f, 0.0f, 4.71f, 5.495f,
                        0.0f, 0.0f, 3.925f };

    // Получаем в каждом кадре локальный ввод
    m_Keyboard.Acquire(TRUE);
    m_Mouse.Acquire(TRUE);
    m_Keyboard.Read();
    m_Mouse.Read();
}
```

В каждом кадре восстанавливаются устройства ввода (в случае если фокус устройства был потерян), и считываются входные данные. Далее игровой процесс может продолжаться если клиент подключен к серверу. Если такой связи не существует, выводится предупреждающее сообщение. Подключение существует только если клиент обработал сообщение **DPN\_MSGID\_CONNECTION\_COMPLETE** и получил идентификационный номер локального игрока.

Для отображения информационного сообщения и ожидания идентификационного номера от сервера используется следующий код:

```
// Обработка экрана подключения
if(g_Connected == FALSE || !m_Players[0].dpnidPlayer) {

    // Отображение сообщения о подключении
    m_Graphics.Clear();
    if(m_Graphics.BeginScene() == TRUE) {
        m_Font.Print("Connecting to server...", 0, 0);
        m_Graphics.EndScene();
    }

    m_Graphics.Display();
    return TRUE;
}
```

Затем происходит разбор пользовательского ввода. Действия игрока отслеживает единственная переменная (**MoveAction**), и каждый разряд в ней представляет отдельное действие (рис. 15.16). Пользовательские действия это перемещение вверх, перемещение вниз, перемещение влево, перемещение вправо и атака. Кроме того, меняется и сохраняется угол камеры (и устанавливается флаг для последующего обновления).



**Рис. 15.16.** Каждый разряд в переменной **MoveAction** используется для определения конкретного действия

Следующий код определяет, какие клавиши нажимает игрок в данный момент, и устанавливает соответствующие разряды переменной **MoveAction**:

```
// Сохраняем перемещение в каждом кадре
if(m_Keyboard.GetKeyState(KEY_UP) == TRUE)
    MoveAction |= 1;

if(m_Keyboard.GetKeyState(KEY_RIGHT) == TRUE)
    MoveAction |= 2;

if(m_Keyboard.GetKeyState(KEY_DOWN) == TRUE)
    MoveAction |= 4;
```



```
if(m_Keyboard.GetKeyState(KEY_LEFT) == TRUE)
    MoveAction |= 8;

// Сохраняем действие атаки
if(m_Keyboard.GetKeyState(KEY_SPACE) == TRUE)
    MoveAction |= 16;

if(m_Mouse.GetButtonState(MOUSE_LBUTTON) == TRUE)
    MoveAction |= 16;

// Вращаем камеру
if(m_Mouse.GetXDelta() > 0) {
    m_CamAngle -= 0.1f;
    CamMoved = TRUE;
}
if(m_Mouse.GetXDelta() < 0) {
    m_CamAngle += 0.1f;
    CamMoved = TRUE;
}

// Обновляем игрока каждые 33 мс (30 раз в секунду)
if(timeGetTime() < UpdateCounter + 33)
    return TRUE;
```

Помните, что обновления игрока ограничены 30 кадрами в секунду, поэтому приведенный выше код возвращает управление, если требуемый период времени еще не прошел.

```
// Устанавливаем флаг, разрешающий перемещение игрока
AllowMovement = TRUE;

// Нельзя перемещаться, если мы размахиваем оружием
if(m_Players[0].State == STATE_SWING)
    AllowMovement = FALSE;

// Нельзя перемещаться, если мы подверглись нападению
if(m_Players[0].State == STATE_HURT)
    AllowMovement = FALSE;
```

Обычно игрокам разрешено перемещаться по миру, но если игрок в данный момент размахивает оружием или подвергся нападению, для него передвижения запрещены. Как видите, чтобы отметить позволено ли игроку выполнять действия, вы используете флаг **AllowMovement**:

```
// Обрабатываем перемещение, если разрешено
if(AllowMovement == TRUE) {

    // Обработка атаки
    if(MoveAction & 16) {
        MoveAction = 0; // Очищаем перемещение
        LastMove = 0; // Очищаем последнее перемещение

        // Отправляем сообщение об атаке -
        // пусть сервер даст сигнал размахивать мечом
        sStateChangeMessage Msg;

        Msg.Header.Type = MSG_STATE_CHANGE;
        Msg.Header.Size = sizeof(sStateChangeMessage);
        Msg.Header.PlayerID = m_Players[0].dpnidPlayer;
```

```

Msg.State = STATE_SWING;
Msg.Direction = m_Players[0].Direction;

// Отправляем сообщение на сервер
SendNetMessage(&Msg, DPNSEND_NOLOOPBACK);
}

```

Если игрок выбрал атаку, надо сконструировать сообщение изменения состояния и отправить это сообщение на сервер. После отправки сообщения изменения состояния пользовательские флаги перемещения очищаются. Обратите внимание, что здесь клиент не меняет свое состояние; когда изменить состояние игрока определяет сервер.

Если игрок не атакует, проверим его действия, чтобы увидеть, не перемещается ли он:

```

// Обработка перемещений локального игрока
if((Dir = MoveAction) > 0 && Dir < 13) {

    // Устанавливаем новое состояние игрока
    // (с временем и направлением)
    EnterCriticalSection(&m_UpdateCS);
    m_Players[0].State = STATE_MOVE;
    m_Players[0].Direction = Angles[Dir] - m_CamAngle + 4.71f;
    LeaveCriticalSection(&m_UpdateCS);

    // Сбрасываем последнее перемещение,
    // если камера передвигалась с момента последнего обновления
    if(CamMoved == TRUE) {
        CamMoved = FALSE;
        LastMove = 0;
    }
}

```

После того, как для игрока установлены его состояние и направление перемещения, функция **Frame** продолжает работу и сбрасывает перемещение камеры (устанавливая флаг **CamMoved** равным **FALSE**). Игрок осуществляет управление относительно угла вида камеры (если игрок нажмет клавишу со стрелкой вверх, он будет идти от камеры). Если вы изменяете угол камеры пока игрок идет, это заставит также измениться и направление движения игрока. Клиент учитывает эти изменения направления игрока когда поворачивается камера.

Теперь функция **Frame** определяет, изменил ли игрок направление перемещения (по сравнению с перемещением в последнем кадре):

```

// Отправляем действие на сервер,
// если последнее перемещение изменилось
if(MoveAction != LastMove) {
    LastMove = MoveAction; // Сохраняем последнее действие
    m_Players[0].Time = timeGetTime();

    // Конструируем сообщение
    sStateChangeMessage Msg;

    Msg.Header.Type = MSG_STATE_CHANGE;
    Msg.Header.Size = sizeof(sStateChangeMessage);
    Msg.Header.PlayerID = m_Players[0].dpnidPlayer;
    Msg.State = STATE_MOVE;
}

```

```
Msg.Direction = m_Players[0].Direction;

// Отправляем сообщение на сервер
SendNetworkMessage(&Msg, DPNSSEND_NOLOOPBACK);
}
```

Когда игрок двигается, клиент отправляет сообщение изменения состояния на сервер. Заметьте, что сообщение изменения состояния отправляется только если перемещение игрока отличается от последнего перемещения, которое он выполнял (записанного в переменной **LastMove**).

Если игрок не двигается, его состояние меняется на остановку (**STATE\_IDLE**), и с помощью показанного ниже кода сообщение изменения состояния отправляется на сервер:

```
} else {
    // Меняем состояние на ожидание
    EnterCriticalSection(&m_UpdateCS);
    m_Players[0].State = STATE_IDLE;
    LeaveCriticalSection(&m_UpdateCS);

    // Отправляем обновление только если во время
    // последнего обновления игрок двигался
    if (LastMove) {
        LastMove = 0;

        sStateChangeMessage Msg;

        Msg.Header.Type = MSG_STATE_CHANGE;
        Msg.Header.Size = sizeof(sStateChangeMessage);
        Msg.Header.PlayerID = m_Players[0].dpnidPlayer;
        Msg.State = STATE_IDLE;
        Msg.Direction = m_Players[0].Direction;

        // Отправляем сообщение на сервер
        SendNetworkMessage(&Msg, DPNSSEND_NOLOOPBACK);
    }
}
```

К этому моменту действия локального игрока зафиксированы и отправлены серверу. Затем обновляются все игроки, визуализируется сцена и действия перемещения сбрасываются для следующего кадра:

```
// Обновляем всех игроков
UpdatePlayers();

// Визуализируем сцену
RenderScene();

MoveAction = 0; // Очищаем данные действий для следующего кадра
UpdateCounter = timeGetTime(); // Сбрасываем счетчик обновлений

return TRUE;
}
```

## Обновление всех игроков

В то время как входные данные игрока обрабатываются в функции **cApp::Frame**, **UpdatePlayers** (вызов которой вы видели в коде из предыдущего раздела) обрабатывает игроков согласно соответствующим им состояниям.

В отличие от серверной функции **UpdatePlayer**, клиентская функция **UpdatePlayer** проста. Клиенту позволено перемещать игроков, основываясь только на их последнем известном местоположении, направлении и времени, прошедшем с момента их последнего обновления.

Помните, что только сервер может сбросить состояния размахивания оружием или получения ранений, таким образом клиент в этой точке ничего не делает, кроме обновления различных анимаций, чтобы показать игроку что происходит:

```
void cApp::UpdatePlayers()
{
    long i;
    float XMove, ZMove, Dist, Speed;
    long Elapsed;

    // Обрабатываем перемещения всех активных игроков
    for(i = 0; i < MAX_PLAYERS; i++) {
        if(m_Players[i].Connected == TRUE) {

            // Получаем время, прошедшее с момента установки состояния
            Elapsed = timeGetTime() - m_Players[i].Time;

            // Обрабатываем состояние перемещения игрока
            if(m_Players[i].State == STATE_MOVE) {

                // Вычисляем дальность перемещения, основываясь
                // на прошедшем времени перемещения
                Speed = (float)Elapsed / 1000.0f * m_Players[i].Speed;
                XMove = (float)sin(m_Players[i].Direction) * Speed;
                ZMove = (float)cos(m_Players[i].Direction) * Speed;

                // Проверяем столкновения при перемещении -
                // нельзя продолжать перемещение, если что-то
                // блокирует путь
                if(m_NodeTreeMesh.CheckIntersect(
                    m_Players[i].XPos,
                    m_Players[i].YPos + 16.0f,
                    m_Players[i].ZPos,
                    m_Players[i].XPos + XMove,
                    m_Players[i].YPos + 16.0f,
                    m_Players[i].ZPos + ZMove,
                    &Dist) == TRUE)
                    XMove = ZMove = 0.0f;

                // Обновляем координаты
                EnterCriticalSection(&m_UpdateCS);
                m_Players[i].XPos += XMove;
                m_Players[i].YPos = 0.0f;
                m_Players[i].ZPos += ZMove;
                m_Players[i].Time = timeGetTime(); // Сброс времени
```

```
        LeaveCriticalSection(&m_UpdateCS);
    }

    // Установка новой анимации, если необходимо
    if(m_Players[i].State == STATE_IDLE) {
        if(m_Players[i].LastAnim != ANIM_IDLE) {
            EnterCriticalSection(&m_UpdateCS);
            m_Players[i].LastAnim = ANIM_IDLE;
            m_Players[i].Body.SetAnimation(
                &m_CharacterAnim, "Idle", timeGetTime() / 32);
            LeaveCriticalSection(&m_UpdateCS);
        }
    } else
        if(m_Players[i].State == STATE_MOVE) {
            if(m_Players[i].LastAnim != ANIM_WALK) {
                EnterCriticalSection(&m_UpdateCS);
                m_Players[i].LastAnim = ANIM_WALK;
                m_Players[i].Body.SetAnimation(
                    &m_CharacterAnim, "Walk",
                    timeGetTime() / 32);
                LeaveCriticalSection(&m_UpdateCS);
            }
        } else
            if(m_Players[i].State == STATE_SWING) {
                if(m_Players[i].LastAnim != ANIM_SWING) {
                    EnterCriticalSection(&m_UpdateCS);
                    m_Players[i].LastAnim = ANIM_SWING;
                    m_Players[i].Body.SetAnimation(
                        &m_CharacterAnim, "Swing",
                        timeGetTime() / 32);
                    LeaveCriticalSection(&m_UpdateCS);
                }
            } else
                if(m_Players[i].State == STATE_HURT) {
                    if(m_Players[i].LastAnim != ANIM_HURT) {
                        EnterCriticalSection(&m_UpdateCS);
                        m_Players[i].LastAnim = ANIM_HURT;
                        m_Players[i].Body.SetAnimation(
                            &m_CharacterAnim, "Hurt",
                            timeGetTime() / 32);
                        LeaveCriticalSection(&m_UpdateCS);
                    }
                }
    }
}
```

Анимация персонажей обновляется только если она отличается от последней известной анимации. Отслеживает последнюю известную анимацию переменная **sPlayer::LastAnim**, а различные макроопределения **ANIM\_\*** определяют, какая анимация воспроизводится.

## Клиент во всем блеске

Тяжелая работа позади! Единственные требования для запущенного клиента — обработка входных данных локального игрока и обновление игроков. Теперь вам осталось нафаршировать ваш проект какой-нибудь трехмерной графикой, и ваша игра почти готова.

Графическая часть клиентского приложения использует графическое ядро для отрисовки в игре различных подключенных игроков. Для визуализации игрового уровня вы используете объект **NodeTree**. Клиент загружает все сетки при инициализации класса приложения. Как упоминалось ранее, все игроки получают назначенные сетки для представления их персонажей и вооружения. Также используются анимации, устанавливаемые согласно различным сообщениям обновления.

Скорость визуализации ограничена 30 кадрами в секунду, и, чтобы обеспечить максимально возможное быстродействие, вы используете пирамиду видимого пространства для визуализации уровня и исключения невидимых персонажей из цикла визуализации.

Завершая описание клиентского приложения, упомяну о том, что вы будете иметь дело еще с несколькими модулями кода приложения, такими как выбор адаптера и подключение к серверу. Полный код вы найдете на прилагаемом к книге CD-ROM (загляните в папку \BookCode\Chap15\Client).

## Заканчиваем с многопользовательскими играми

Обсужденные в этой главе игровой сервер и клиент достаточно мощны (независимо от того, насколько примитивными они могли вам показаться). После небольшой переработки вы можете их подстроить для точного соответствия потребностям вашего игрового проекта. Если вы хотите создавать игры, могущие одновременно поддерживать тысячи игроков, я рекомендую вам исследовать многосерверные конфигурации, использующие лобби-серверы и серверы подключений. Для огромного игрового мира вам может понадобиться 100 компьютеров, чтобы разместить единую постоянную игровую сессию. Ничего удивительного!

Для дальнейшего исследования темы многопользовательских игр я настоятельно рекомендую вам приобрести книгу Тодда Баррона «Программирование многопользовательских игр». За дополнительной информацией о ней обратитесь к Приложению А, «Список литературы».

Кроме того, вы можете отправиться на передовую многопользовательских игр, загрузив исходные коды Quake и Quake II (две игры id Software, Inc, которые помогли революционизировать многопользовательский жанр). Quake и Quake II являются замечательным действующим примером многопользовательских игр, и, прочитав исходный код этих двух игр, вы хорошо изучите использование сетевых возможностей в играх. Чтобы загрузить исходный код Quake и Quake II, направьте ваш браузер (или FTP-клиент) по адресам <ftp://ftp.idsoftware.com/idstuff/source/q1source.zip> и <ftp://ftp.idsoftware.com/idstuff/source/quake2.zip>.

### Программы на CD-ROM

На прилагаемом к книге CD-ROM расположены два проекта, использующие многопользовательские возможности, показанные в этой главе. В каталоге \BookCode\Chap15 вы найдете следующие программы:

**Server** — многопользовательский игровой сервер, позволяющий подключение и одновременную игру восьми игроков. Местоположение: \BookCode\Chap15\Server\.

**Client** — клиентское приложение, позволяющее игроку подключиться к удаленному серверу и присоединиться к семи другим игрокам для многопользовательской игры. Для возможности игры необходимо, чтобы был запущен сервер. Местоположение: \BookCode\Chap15\Client\.

# **Часть IV**

## **Завершающие штрихи**

### **Глава 16      Объединяем все вместе в законченную игру**





# Глава 16

## Объединяем все вместе в законченную игру

Весь ваш тяжелый труд окупается! В этой главе вы соберете все части игры вместе. Если вы последовательно читали эту книгу, у вас уже полностью разработаны базовый движок, карты, персонажи, предметы и скрипты. Осталось совместить все это воедино и получить игру! В этой главе вы постройте небольшую игру, которую я разработал специально для этой книги.

В главе будут рассмотрены следующие темы:

- Проектирование простой игры.
- Сборка игровых компонентов.
- Программирование игры.

### Проектирование простой игры

Пришло время начать создание примера законченной игры из этой книги — The Tower. Хотя название не слишком интригующее, проект действительно объединяет вместе все фрагменты, необходимые для законченной игры. Цель The Tower — показать каждый компонент на его правильном месте, включая технологический движок, управление картами и уровнями, контроль персонажей и скрипты.

Создание игры начинается с проектирования. Вам необходимо написать сценарий, создать уровни, разработать персонажей, спроектировать предметы и изобрести заклинания. Чтобы помочь определиться с контекстом, я буду описывать процесс проектирования в виде короткого рассказа, описывающего, что происходит после того, как игрок попадает в игру. История объясняет, откуда игрок попал в мир игры, что задает настроение и т.д.

### Написание сценария игры

Часть исследования новой игры состоит из изучения буклета с инструкцией, знакомящего с управлением игрой, персонажами и предшествующей началу

игры историей. Верно, вам нужен не только сценарий для игры, но и небольшая история, описывающая события, происходившие до начала игры.

---

**ПРИМЕЧАНИЕ**

В своей игре, вместо того, чтобы писать предысторию в буклете с инструкцией, вы можете включить ее в саму игру. Например, когда игрок в первый раз начинает игру, вы можете сразу, до начала игры, показать предысторию. Это гарантирует, что все игроки будут знать завязку сюжета, независимо от того, прочитали ли они буклет.

---

Игровая предыстория, такая как показана ниже, задает настрой для игры (или, как минимум, для начала игры), знакомит игрока с его альтер-эго и переносит игрока прямо в сюжет. Для The Tower я написал следующую предысторию, позволяющую игроку узнать свою роль и ситуацию, в которой он находится в начале игры.

Подошло время странствий. После краткой остановки в странной деревне Дансберри наш герой снова отправляется исследовать холмистые равнины Восточных Земель. Ранее он трижды путешествовал по этим Землям — они быстро стали его любимым местом приключений и не зря. Каждый раз нашего героя ждали волнующие приключения, невообразимые сокровища и самые загадочные люди из тех, с которыми он когда-либо имел удовольствие встречаться.

Надеясь на свой разум и оставив солнце за спиной, наш одинокий искатель приключений отправился вперед. Если ничто не задержит, он должен достигнуть деревни Грандер перед рассветом. Там он остановится отдохнуть, перед тем как идти в центральные районы Земель, где есть надежда найти возможность совершить великие подвиги.

Внезапный оглушительный удар грома вымел мечты о славных завоеваниях из его головы. День заканчивался — солнце село. Подул пронзительный ветер, нагоняя темные, клубящиеся облака. Ночь становилась все темнее, ветер крепчал каждую секунду и росли облака. Тяжелые раскаты грома разлились в воздухе, а краткие вспышки молний освещали равнины мертвенно-голубым цветом.

Зарождающийся на равнине шторм быстро увеличивал свою силу. С последним оглушительным ударом грома облака начали изливаться свое содержимое. Яростный проливной дождь размыл землю, делая дальнейшее путешествие невозможным. Нашему искателю приключений необходимо найти убежище, иначе шторм будет угрожать его жизни.

Шатаясь он достиг небольшой группы деревьев, среди которых и поставил свою палатку. Из последних сил измученный искатель приключений закрыл полог наскоро установленной палатки и зарылся в постель из мокрого сена и ткани. Скоро он впал в забытие, перешедшее в глубокий беспокойный сон. Видения отвратительных чешуйчатых демонов заполнили его сны. Великая башня вырисовывалась над ним, отбрасывая черную тень на окружающие земли. Перед героем появилась маленькая деревня — путь ясен; деревня призывает его вперед.

Как видите, хотя история для нашего героя начинается хорошо, обстоятельства переворачивают все вверх дном. Начинается загадочный шторм и вынуждает искать убежище. По завершении этой части истории начинается игровой процесс. Предыдущая история не объясняет, что происходит с героем, когда в своих снах он входит в деревню. Теперь вы должны определить цель сценария, то есть, что игрок (как главный герой игры) должен делать, начиная с этого места истории.

## Цель игры The Tower

История из предыдущего раздела помогает установить настрой для The Tower и объясняет, как игрок очутился в маленькой деревне, расположенной рядом с темной зловещей башней. Цель игрока в The Tower — освободить проклятую деревню от злых демонов, населяющих расположенную рядом башню. Оказывается, жители деревни пойманы в ловушку и должны быть один за другим принесены в жертву зловещему повелителю демонов башни. Задача игрока — пойти в башню и уничтожить злобных созданий.

По ходу игры игрок может использовать различное оружие и заклинания. Убийство монстров увеличивает очки опыта игрока. После набора определенного количества очков увеличивается уровень опыта и способности игрока. По достижении нового уровня игрок также получает возможность использовать новые изученные заклинания.

Эти предметы и заклинания пригодятся, потому что игрок должен прорубиться через пять заполненных монстрами уровней (включая деревню, где игрок может восстановить здоровье и купить экипировку). Каждый уникальный уровень кратко демонстрирует, что вы можете достичь, используя созданные в книге компоненты. Итак, пришло время двигаться дальше и посмотреть, как проектируются эти уровни.

## Разработка уровней

Игра The Tower состоит из пяти игровых уровней (также называемых сценами) — деревня, мост, нижний уровень башни, балкон башни, и комната Злого Лорда. Каждый уровень представлен заранее визуализированным растровым изображением задника, точно так же, как это было показано в главе 9, «Смешивание двухмерной и трехмерной графики». Во всех игровых уровнях есть одно растровое изображение, разделенное на шесть небольших текстур. Также у каждого уровня есть упрощенная сетка, используемая для визуализации буфера глубины данного уровня.

Первая сцена, деревня, показана на рис. 16.1. Деревня — это начальный пункт приключения и то место, где игрок может найти исцеление и купить предметы. В начале игры, когда прибывает игрок, в деревне присутствует только единственный монстр. Как только он будет повержен, деревня становится достаточно безопасной, и ее жители выходят из укрытий.



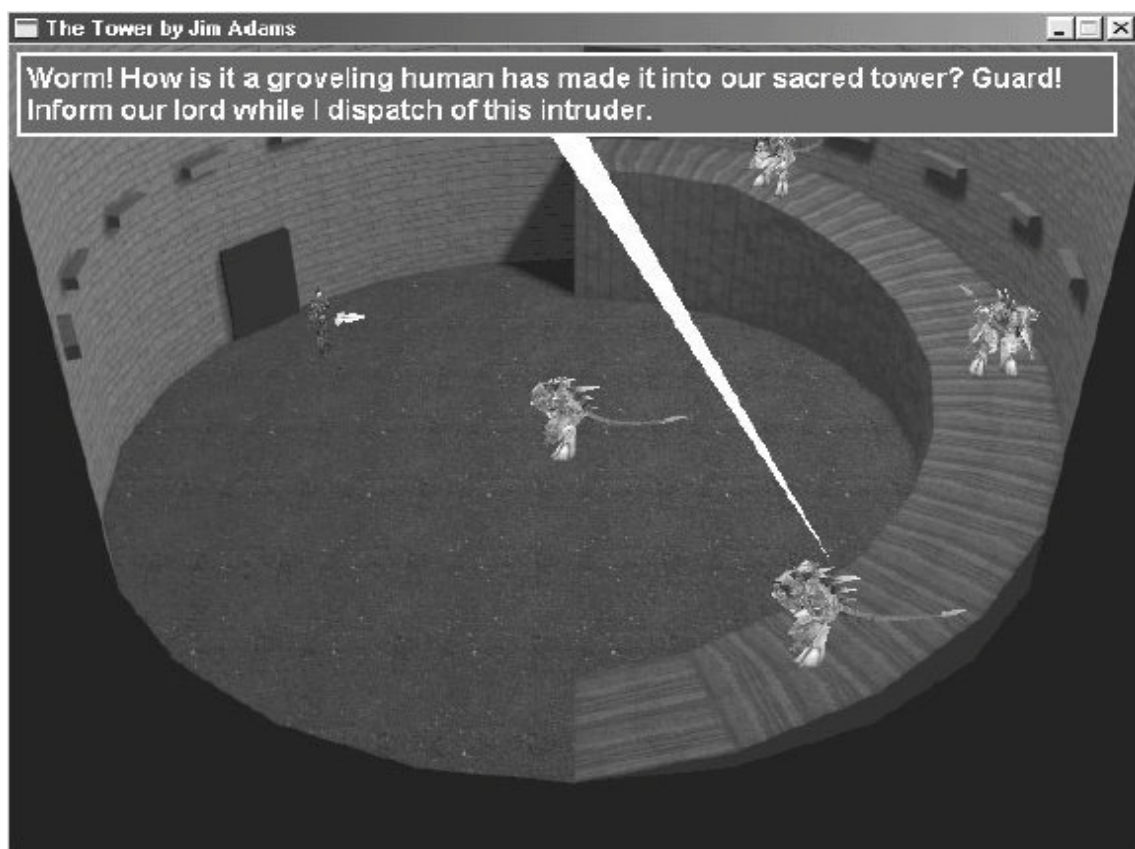
*Рис. 16.1. Уровень деревни, заполненный местными жителями, это то место, где игрок начинает свое приключение*



*Рис. 16.2. На мосту к башне встречаются опасности. На рисунке два монстра готовятся к убийству*

На рис. 16.2 показан второй уровень игры, мост. Во время первого посещения моста игроком деревенская стража блокирует проход через мост. Как только стражник покинет свой пост, игрок может свободно перейти мост и попасть в башню. После того как стражник вернулся в деревню на уровне случайным образом размещаются монстры.

Третий уровень игры, показанный на рис. 16.3, — это нижний этаж башни. Когда игрок входит в башню первый раз, он видит двух стражей. Один страж атакует, а другой идет предупредить своего властелина о вторжении игрока. В последующие посещения нижнего этажа башни там случайным образом размещается несколько атакующих стражников.



*Рис. 16.3. Нижний этаж башни постоянно патрулируют два или более стражников. На рисунке один стражник отдает команду другому*

На рис. 16.4 вы видите четвертый уровень, балкон башни. Игрок должен пройти по этому балкону, чтобы достичь последнего уровня. Здесь игрока ожидает битва против Гранита, стражника, который был магически преобразен властелином за позднее предупреждение о появлении игрока.

Последний уровень игры, комната Злого Лорда (рис. 16.5), — это место где игра подходит к завершению. Внутри круглых стен этого последнего уровня живет Злой Лорд, предводитель демонов, ответственный за наложенное на деревню проклятие. Чтобы выиграть игрок должен убить Злого Лорда.





*Рис. 16.4. Балкон башни является логовом мощного демона Гранита. Будьте осторожны; у него в запасе есть пара сюрпризов и он очень хочет побыстрее избавиться от игрока*



*Рис. 16.5. Злой Лорд живет в ожидании игроков, посещающих его внутреннее святилище. На рисунке Злой Лорд произносит заклинание, повышающее его мощь*

Хотя количество персонажей ограничено, пример игры замечательно демонстрирует работу с персонажами. В следующем разделе вы увидите как определить игровые персонажи, а в разделе «Управление персонажами», далее в этой главе, вы увидите как эти персонажи программируются в проекте.

## Определение персонажей

В игре The Tower есть восемь типов персонажей, включая игрока (в игре используется несколько экземпляров одного и того же типа персонажа). Эти восемь типов персонажей описаны в таблице 16.1. Чтобы разобраться в этих определениях, воспользуйтесь следующим списком обозначений:

- **Class.** Класс персонажа.
- **HP.** Очки здоровья. Максимальное количество очков здоровья персонажа.
- **MP.** Очки маны. Максимальное количество очков маны персонажа.
- **Exp.** Количество опыта, приобретаемого при убийстве персонажа.
- **Gold.** Количество имеющегося у персонажа золота (оно выпадает, когда персонажа убивают).
- **Atk.** Способность атаковать.
- **Def.** Способность защищаться.
- **Agl.** Проворность.
- **Res.** Способность сопротивляться магии.
- **Mnt.** Интеллект персонажа.
- **ToH.** Меткость персонажа.
- **Speed.** Скорость передвижения, измеряемая в единицах за секунду.
- **Range.** Дистанция, на которой персонаж может атаковать, измеренная в единицах.
- **Charge.** Скорость перезарядки персонажа. Измеряется в единицах за секунду. Персонаж может атаковать или произносить заклинание только когда его заряд достигнет 100.
- **Atk%.** Процент, определяющий вероятность того, что персонаж атакует приблизившегося игрока оружием.
- **Mag%.** Процент, определяющий вероятность того, что персонаж атакует приблизившегося игрока магическим заклинанием.
- **Spells.** Заклинания, которые знает персонаж.



Таблица 16.1. Персонажи игры The Tower

№	Название	Описание
0	Игрок	Персонаж, представляющий игрока. Приведенная ниже статистика относится к началу игры. Class 1, HP:25, MP:10, Atk:6, Def:3, Agl:2, Res:2, Mnt:2, ToH:800, Speed:150, Range:16, Charge:40. Игрок может выучить заклинания с 0 по 4, достигая каждого нового уровня опыта, начиная с уровня 2. Уровень опыта повышается при наборе 10, 30, 80, 150, 350 и 500 очков опыта.
1	Демон-пеон	Слабый демон. Class 2, HP:14, Exp:10, Gold:10, Atk:8, Def:1, Agl:1, Res:0, Mnt:0, ToH:700, Speed:64, Range:8, Charge:30.
2	Демон-страж	Более сильный вариант пеона. Class 2, HP:28, MP:20, Exp:50, Gold:20, Atk:10, Def:5, Agl:3, Res:2, Mnt:2, ToH:800, Speed:64, Range:10, Charge:30, Atk%:80, Mag%:20, Spells:0, 2.
3	Гранит	Этот рожденный магией каменный монстр представляет опасность! Class 3, HP:70, MP:100, Exp:200, Gold:0, Atk:20, Def:15, Agl:5, Res:10, Mnt:10, ToH:800, Speed:16, Range:16, Charge:15, Atk%:80, Mag%:20, Spells:4.
4	Злой Лорд	Король хаоса, этот демон — цель игрока, пойманного в ловушку в проклятой деревне. Class 4, HP:150, MP:200, Exp:1000, Gold:500, Atk:25, Def:25, Agl:10, Res:40, Mnt:50, ToH:900, Speed:16, Range:16, Charge:40, Atk%:70, Mag%:30, Spells:0, 5, 6.
5	Старейшина	Деревенский староста. Он исцеляет вас всякий раз, когда вы разговариваете с ним.
6	Стражник	Деревенский стражник. Блокирует проход через мост к башне.
7	Крестьянин	Деревенский лавочник. Продает магические предметы и целебные эликсиры.

В качестве примера использования описания персонажа и списка обозначений возьмем персонаж с номером 3. Гранит (имя персонажа) относится к классу 3, имеет 70 очков здоровья, 100 очков маны, 200 очков опыта (они даются игроку, когда тот убьет Гранита), у него нет золота, и он обладает следующими способностями:

Атака:	20	Скорость передвижения:	16 единиц в секунду
Защита:	15	Дальность атаки:	16 единиц
Ловкость:	5	Скорость зарядки:	15 единиц в секунду
Сопrotивляемость магии:	10	Вероятность атаки оружием:	80%
Интеллект:	10	Вероятность использования магии:	20%
Меткость:	800	Известные заклинания:	4 (земляной вал)

Чтобы лучше прочувствовать смысл определений персонажей я рекомендую загрузить программу MCLEdit и использовать ее для редактирования файла Game.mcl (все это находится на прилагаемом к книге CD-ROM; загляните в каталог \BookCode\Chap12\MCLEdit). В списке есть только один персонаж, на котором действительно надо остановить внимание, — это персонаж с номером 0, сам игрок.

Статистика персонажа, приведенная в таблице 16.1 (и в главном списке персонажей, используемом в демонстрационной игре — смотрите \BookCode\Chap16\Data\Game.mcl) представляет сведения для начала игры.

---

<b>ПРИМЕЧАНИЕ</b>	Вы можете редактировать главный список персонажей (MCL), используя программу MCL Editor, которая была описана в главе 12, «Управление игроками и персонажами».
-------------------	--

---

Когда игрок убивает монстра, очки опыта игрока немного увеличиваются (в зависимости от имеющихся у монстра очков опыта), и когда игрок достигнет определенного уровня очков опыта, его статистика увеличивается на следующие значения:

- Максимальное количество очков здоровья: +10.
- Максимальное количество очков маны: +10.
- Атака: +2.
- Защита: +1.
- Ловкость: +1.
- Сопrotивляемость магии: +1.
- Меткость: +5.

Помимо увеличения статистики, игрок узнает новые заклинания, в зависимости от достигнутого уровня опыта. Ниже приведены значения очков опыта по достижении которых происходит это «повышение уровня» опыта, вместе с изучаемыми заклинаниями:

- 10 очков опыта для уровня 2. Игрок изучает заклинание огненного шара (заклинание 0).
- 30 очков опыта для уровня 3. Игрок изучает заклинание льда (заклинание 1).
- 80 очков опыта для уровня 4. Игрок изучает заклинание исцеления (заклинание 2).
- 150 очков опыта для уровня 5. Игрок изучает заклинание телепортации (заклинание 3).
- 350 очков опыта для уровня 6. Игрок изучает заклинание земляного вала (заклинание 4).
- 500 очков опыта для уровня 7 (финальный уровень). Новых заклинаний нет. Заметьте, что 7 уровень — это последний

достижимый для игрока уровень опыта. К этому моменту игрок должен быть достаточно силен, чтобы победить Злого Лорда в конце игры.

## Распределение персонажей

В процессе разработки примера игры необходимо каждому персонажу назначить уникальный идентификационный номер. Например, игроку будет назначен идентификатор 0, в то время как для деревенского старейшины будет использоваться идентификатор 1. Благодаря назначению этих идентификационных номеров скриптовый движок будет знать, какого персонажа использовать для выполнения определенного действия, такого как отображение диалогов или отслеживание изменяющихся по ходу игры флагов. Вот эти назначенные идентификационные номера:

- 0: персонаж игрока.
- 1: деревенский старейшина.
- 2: деревенский стражник.
- 3: деревенский лавочник.
- 4: монстр в деревне в начале игры.
- 5: демон в башне, бегущий предупредить Злого Лорда.
- 6: Гранит, каменный демон.
- 7: Злой Лорд.

Единственные персонажи, которых нет в этом списке распределения идентификационных номеров, это монстры, с которыми игрок сталкивается на протяжении всей игры. Вы назначаете этим монстрам идентификационные номера от 256 и далее — нет никакой причины для того, чтобы здесь заранее определять эти номера. Порядок нумерации прост — прибывший первым будет первым обслужен и получит свой идентификационный номер монстра.

## Создание предметов и заклинаний

В игре The Tower восемь предметов, и все они перечислены в таблице 16.2. Несколько предметов продаются; остальные игрок получает от других персонажей. Прогаваемые предметы перечислены с ценой (указывается числом, за которым идут буквы *gp*), и их можно приборести на уровне с деревней.

В примере игры вы используете семь заклинаний пронумерованных от 0 до 6. Все они перечислены и описаны в таблице 16.3. Для исследования специфики каждого заклинания вы можете воспользоваться программой Master Spell List Editor из главы 12. Заклинания игры содержатся в файле `Game.msl` (он расположен на прилагаемом к книге CD-ROM; загляните в `\BookCode\Chap16\Data\Game.msl`).

Таблица 16.2. Предметы игры The Tower

№	Название	Описание
0	Золото	Валютой в игре The Tower являются золотые монеты.
1	Меч	Оружие, увеличивающее наносимый атакующим урон на 15%. Игрок начинает игру с мечом.
2	Магический меч	Сильный меч, увеличивающий наносимый атакующим урон на 50%. Цена: 100gr.
3	Кожаный доспех	Слабый обшитый кожей доспех, уменьшающий повреждения от атаки на 10%. Вы получаете его от деревенского стражника.
4	Магическая броня	Превосходная броня из зачарованного металла. Уменьшает ущерб от атаки на 50%. Цена: 100gr.
5	Малый щит	Небольшой слабый щит, уменьшающий ущерб от атаки на 5%. Вы получаете его от деревенского стражника.
6	Магический щит	Превосходный щит из шкуры красного дракона. Уменьшает ущерб от атаки на 20%. Цена: 100gr.
7	Целебный эликсир	Одна доза этого эликсира добавляет вам 50 очков здоровья. Цена: 10gr.

Таблица 16.3. Заклинания игры The Tower

№	Название	Описание
0	Огненный шар	Швыряет шар огня в одиночного врага для нанесения легких повреждений. Цена: 5, Исцеление: 4, Эффект: Изменение здоровья -10, Цель: одиночная, Радиус эффекта: 32, Дальность: 512.
1	Лед	Покрывает жертву льдом, нанося средние повреждения. Цена: 10, Двойная опасность: 4, Эффект: Изменение здоровья -20, Цель: одиночная, Радиус эффекта: 40, Дальность: 512.
2	Исцеление	Восстанавливает 50 очков здоровья заклинателя. Цена: 8, Эффект: Изменение здоровья +50, Цель: на себя.
3	Телепортация	Телепортирует игрока в ближайший город. Цена: 10, Эффект: Телепортация, Цель: на себя.
4	Земляной вал	Пробуждает подземные силы, громоздящие куски скал. Цена: 10, Исцеление: 3, Эффект: Изменение здоровья -30, Цель: одиночная, Радиус эффекта: 40, Дальность: 512.
5	Контузия	Огромный выброс силы, наносящий серьезные повреждения. Цена: 20, Эффект: Изменение здоровья -40, Цель: область, Радиус эффекта: 1024, Дальность: 1024.
6	Сила Зла	Заклинатель увеличивает свою мощь, оказавшись заключенным в шар тьмы. Цена: 10, Эффект: установка статусов 10832, Цель: на себя.

Чтобы лучше разобраться с заклинаниями в таблице 16.3, посмотрите следующий список условных обозначений:

- **Цена.** Количество очков манны, необходимое чтобы произнести заклинание.
- **Исцеление.** Заклинание может исцелять персонажей с указанным классом. Если указан исцеляемый класс, направление заклинания на такой персонаж исцеляет его на половину предполагаемых повреждений.
- **Двойная опасность.** Подобно исцелению, здесь указывается класс персонажей, которому данное заклинание наносит ущерб в два раза больше, чем всем остальным. Например, заклинание льда наносит двойной ущерб огненным монстрам, таким как Злой Лорд.
- **Эффект.** Эффект, оказываемый заклинанием на указанный персонаж. В игре The Tower используются четыре эффекта: изменение здоровья, телепортация, снятие статусов и установка статусов. За каждым эффектом (корме телепортации) следует число, представляющее сначение модификатора. Для изменения здоровья это количество вычитаемых или прибавляемых цели очков здоровья. Для снятия и установки статусов это битовая маска для изменения флагов состояния.
- **Цель.** Это тип цели заклинания, которой может быть отдельный персонаж (одиночная), сам персонаж произносящий заклинание (на себя) или часть уровня (область).
- **Радиус эффекта.** Это радиус в пределах которого эффект заклинания поражает персонажи.
- **Дальность.** Максимальное расстояние, которое заклинание может преодолеть, чтобы поразить указанного персонажа. Произносящий заклинание персонаж должен находиться не дальше указанного расстояния от намеченной цели, чтобы заклинание подействовало.

---

<b>ПРИМЕЧАНИЕ</b>	Вы можете просматривать и редактировать заклинания игры The Tower с помощью программы Master Spell List Editor, которая рассматривалась в главе 12.
-------------------	---

---

Некоторые из представленных выше заклинаний являются уникальными возможностями отдельных персонажей. Например, заклинание силы зла использует только Злой Лорд — он направляет его на себя чтобы увеличить свою мощь (особым образом устанавливая дополнительные статусы и увеличивая скорость). Обратитесь к главному списку персонажей игры (\BookCode\Chap16\Data\Game.mcl), чтобы определить какой персонаж какие заклинания знает.

Вот и все о содержании игры, заклинаниях и персонажах, так что теперь вы можете обратить свое внимание на игровые скрипты.

## Разработка скриптов

Вы управляете всем игровым содержимым The Tower, таким как диалоги, посредством использования скриптов. Здесь применяется скриптовая система Mad Lib Scripts, которая была рассмотрена в главе 10, «Реализация скриптов». Единый шаблон действий, `Game.mla` (смотрите `\BookCode\Chap16\Data\Game.mla`) содержит набор действий, которые могут оказаться полезными для ваших проектов. Игровой шаблон действий занимает более 200 строк и слишком велик, чтобы быть приведенным здесь, поэтому я настоятельно рекомендую вам открыть его в текстовом редакторе и обращаться к нему по ходу чтения данного раздела.

Шаблон действий разделен на следующие шесть групп действий:

- **Поток исполнения скрипта.** Подобно обычной программе, исполнение скрипта начинается с его начала. Поток исполнения продолжается, пока не будет достигнут конец скрипта. Скрипты также используют условные проверки `if...then` (контролирующие состояние внутренних флагов и переменных) для изменения потока исполнения скрипта.
- **Управление персонажами.** Сюда входит добавление, удаление и перемещение персонажа, а также установка параметров искусственного интеллекта персонажа.
- **Управление предметами.** Это проверки, существует ли предмет, и добавление и удаление предметов в имущество персонажа.
- **Барьеры и триггеры.** Эта группа действий позволяет вам добавлять, активировать и удалять триггеры карты.
- **Звуки и музыка.** Вы можете воспроизводить различные звуковые эффекты и песни, используя данную группу действий.
- **Разное.** Группа действий не подходящих ни к одной из перечисленных выше категорий.

Представленные действия вы используете для конструирования игровых скриптов. Когда скрипты сконструированы вы применяете их для управления потоком исполнения игры. Вы запускаете используемые в игре скрипты шестью различными способами — когда игрок разговаривает с другим персонажем, когда игрок касается триггера на карте, когда персонаж достигает конечной точки назначенного ему маршрута, при переходе на другой уровень, в начале сражения и по завершении сражения.

Скрипты, которые вызываются когда игрок обращается к другому персонажу, вы именуете в соответствии с идентификационным номером персонажа перед которым ставится слово `char`. Например, у персонажа с номером 2 есть файл скрипта с именем `char2.mls`, исполняемый каждый раз, когда игрок щелкает по этому персонажу кнопкой мыши.

Вы размещаете триггеры карты на каждом уровне используя действия скрипта. Когда триггер срабатывает, запускается связанный с ним скрипт.

Вы составляете имена триггеров карты из слова `trig` и следующего за ним идентификационного номера триггера — таким образом, триггер с номером 4 использует скрипт из файла с именем `trig4.mls`.

Для скриптов входа на уровень используется слово `scene`, за которым идет номер, назначенный карте уровня. Например, когда персонаж попадает на карту с номером 4, выполняется скрипт `scene4.mls`.

Для завершающих трех методов запуска скрипта используются трехбуквенные имена файлов скрипта, за которыми идет идентификационный номер персонажа или карты уровня. Для скрипта завершения маршрута вы используете имя `eor` (*end of route*) за которым следует идентификационный номер персонажа. К примеру, когда персонаж с номером 2 достигает конечной точки маршрута, выполняется скрипт `eor2.mls`.

---

<b>ПРИМЕЧАНИЕ</b>	<i>Скрипт завершения маршрута (end-of-route script) обрабатывается когда персонаж достигает последней точки назначенного ему маршрута.</i>
-------------------	--

---

Для имен файлов скриптов начала сражения используется сокращение `soc` (*start of combat*), за которым следует номер карты уровня. То же самое применимо и к скриптам завершения сражения, только используется `eoc` (*end of combat*) — например, `eoc3.mls` исполняется когда завершается сражение на карте уровня с номером 3.

Держа в уме эти шесть методов именования файлов скриптов, взгляните на приведенный ниже список скриптов, используемых в игре *The Tower* (вы найдете все эти скрипты на CD-ROM):

- **Char1.mls, Char2.mls, Char3.mls, Char6.mls и Char7.mls.** Эти скрипты исполняются каждый раз, когда игрок щелкает левой кнопкой мыши по персонажу. Персонажи 1, 2 и 3 это жители деревни, а персонажи 6 и 7 это Гранит и Злой Лорд.
- **SOC1.mls, SOC2.mls, SOC3.mls, SOC4.mls и SOC5.mls.** Это скрипты начала сражения для каждого уровня. Они просто запускают воспроизведение одной из трех назначенных звуковых тем.
- **EOC1.mls, EOC2.mls, EOC3.mls, EOC4.mls и EOC5.mls.** Скрипты завершения сражения обычно возвращают музыку к исходной теме уровня.
- **EOR0.mls, EOR4.mls и EOR5.mls.** Только три персонажа игры ходят по заданному маршруту — игрок в первом уровне игры, демон, атакующий игрока в начале игры и стражник, убегаящий предупредить Злого Лорда.
- **Scene1.mls, Scene2.mls, Scene3.mls, Scene4.mls и Scene5.mls.** Каждая сцена начинается с воспроизведения музыки и расстановки всех относящихся к данному уровню персонажей.

- **Trig1.mls, Trig2.mls, Trig3.mls, Trig4.mls, Trig5.mls, Trig6.mls, Trig7.mls** и **Trig8.mls**. Вы используете триггеры карты исключительно для перемещения игрока с одного уровня на другой, всякий раз когда игрок пытается покинуть заданный уровень.

Большинство скриптов очень просты. Взгляните, например, на скрипт **trig2.mls**:

```
Set character id=(*0*) direction to (*0.000000*)
Teleport character id=(*0*) to map (*1*) at
    (*100.000000*) (*0.000000*) (*-170.000000*)
```

Цель **trig2.mls**, размещенного на второй сцене (мост), заключается в телепортации персонажа на первую карту (деревня) и изменении направления игрока. Чтобы увидеть более сложный скрипт, взгляните на **scene4.mls**, выполняемый когда игрок попадает на четвертый уровень:

```
// (*Сохраняем номер сцены*) //
Set variable (*1*) to (*4*)
-----
// (*Воспроизводим музыку сцены *) //
Play music (*1*)
-----
// (*Добавляем триггеры телепортации *) //
Add triangle trigger id=(*6*) at
    (*-177.000000*) (*200.000000*) (*-144.000000)
Add triangle trigger id=(*7*) at
    (*177.000000*) (*200.000000*) (*210.000000)
-----
// (*Добавляем Гранита, если он еще не убит *) //
if flag (*8*) equals (*FALSE*) then
    Add character id=(*6*) definition=(*3*) type=(*NPC*) at
        XPos=(*170.000000*) YPos=(*0.000000*) ZPos=(*-60.000000*)
        direction=(*3.925000*)
    Set character id=(*6*) AI to (*Stand*)
EndIf
```

Хотя он и значительно длиннее, чем другие игровые скрипты, **scene4.mls** исключительно прост. Скрипт начинается подобно всем другим скриптам сцен — с сохранения номера карты сцены в переменной с номером 1 и воспроизведения назначенной для уровня музыки. Затем на сцене размещаются два триггера, телепортирующие игрока обратно на первый этаж башни и в покои Злого Лорда.

В завершение скрипта проверяется флаг с номером 8 и, если его значение **FALSE**, к уровню добавляется персонаж. Этот персонаж, Гранит, является персонажем номер 3 в главном списке персонажей. В игровом движении Граниту назначается идентификационный номер персонажа 6. Сперва Гранит помечается как **NPC** (независимый персонаж) и просто стоит и ждет, пока игрок не заговорит с ним.

Когда с ним заговорят, обрабатывается скрипт Гранита — обмен несколькими словами между Гранитом и игроком, после чего тип персонажа Гранита меняется на **Monster**. Когда игрок побеждает Гранита, скрипт завершения сражения присваивает флагу 8 значение **TRUE**, и та часть



скрипта `scene4.mls`, которая добавляет Гранита на карту, будет пропущена, когда игрок снова вернется на сцену с номером 4. Изобретательно, не так ли?

В разделе «Обработка скриптов», далее в этой главе, вы увидите как обрабатываются скрипты в игре The Tower. А сейчас перейдем к определению того, как игрок будет взаимодействовать с игрой.

## Определение управления

Игрок взаимодействует с игрой The Tower используя клавиатуру и мышь. Работая с главным меню (рис. 16.6) игрок использует мышь для выбора одного из пунктов. В главном меню доступны следующие пункты:

- **New Game.** Выберите данный пункт, чтобы начать новую игру.
- **Back to Game.** Возврат к идущей в данный момент игре.
- **Load Game.** Загрузка и продолжение ранее сохраненной игры.
- **Save Game.** Сохранение идущей в данный момент игры.
- **Quit.** Выход из игры.



*Рис. 16.6. Главное меню позволяет игроку начать новую игру, вернуться к идущей игре, загрузить сохраненную игру и выйти из текущей игры*

Чтобы выбрать пункт меню, игрок помещает указатель мыши над одним из отображенных пунктов и нажимает левую кнопку мыши для его выбора. В процессе игры управление несколько более хитрое.

Игрок использует клавиши управления курсором для перемещения персонажа и мышь, чтобы выбирать персонаж, который будет целью заклинания или атаки. Нажатие клавиши перемещения курсора вверх вызывает движение игрока вперед, а клавиши перемещения курсора влево и вправо поворачивают его. Наведите указатель мыши на находящегося вблизи монстра и щелкните левой кнопкой, чтобы атаковать его. Заметьте, что игрок должен быть достаточно близко к монстру, чтобы иметь возможность его атаковать. Щелчок левой кнопкой по NPC начнет «разговор» с этим персонажем. Игроку не надо приближаться вплотную к персонажу для разговора — задачу выполняет простой щелчок по персонажу, расположенному в любом месте экрана.

Нажатие цифровых клавиш от **1** до **5**, в то время как указатель мыши находится над персонажем (не являющимся NPC), приводит к произнесению заклинания, целью которого является данный персонаж. Нажатие клавиши **1** вызывает заклинание огненного шара, нажатие клавиши **2** вызывает заклинание льда, нажатие **3** — заклинание исцеления, **4** вызывает телепортацию и **5** вызывает земляной вал. У заклинаний **3** и **4** целью может быть только игрок, так что не имеет значения, на какого персонажа вы указываете, произнося эти заклинания, их эффект всегда будет действовать на игрока. Заметьте, что произносить можно только известные заклинания, а чтобы определить, какие заклинания знает игрок, следует открыть окно состояния персонажа.

Щелчок правой кнопкой мыши в ходе игрового процесса приводит к открытию окна состояния персонажа. Чтобы использовать предмет, включить его в экипировку или удалить из нее, следует щелкнуть по предмету левой кнопкой мыши. Повторный щелчок правой кнопкой закрывает окно состояния. В нижнем правом углу окна состояния вы можете увидеть числа от 1 до 5, представляющие известные заклинания.

Чтобы выйти из игры и вернуться к главному меню нажмите в ходе игры клавишу **Esc**. Если вы разговариваете с персонажем, щелчок левой кнопкой мыши или нажатие пробела продолжают общение; если вы торгуете с лавочником, щелкните левой кнопкой мыши по покупаемому предмету или правой кнопкой мыши для выхода из окна.

## Планирование хода игры

Спроектировав все отдельные аспекты, пришло время собрать их в завершенную игру. Игра линейна — все происходящее в игре заранее запланировано. У игрока есть прямой путь от начала игры к ее концу, в основном это обусловлено крошечным размером игры.

Игра начинается с прихода игрока в деревню. Сказав себе несколько слов, игрок мельком замечает демона, идущего через поселение. Похоже,

этой ночью будет жертвоприношение, и демон направлен чтобы проводить несчастную душу к месту ее гибели. Испуганный и заинтригованный игрок начинает разговор, только для того, чтобы быть атакованным монстром.

После того, как игрок побеждает мерзкого демона, жители деревни чувствуют себя достаточно безопасно, чтобы выйти из укрытий и поблагодарить игрока за его героический поступок. Кажется, сельские жители полагают, что игрок, это спаситель из старой легенды — легенды, в которой освободитель избавляет их от проклятия, заточившего в ловушку всех жителей близлежащих земель (в основном это деревня и близлежащая башня).

Чтобы не дать пропасть хорошим людям, игрок следует к находящейся на востоке башне. На этом пути его останавливает деревенский стражник, не дающий игроку пересечь единственный ведущий к башне мост. Выполняя свои обязанности, стражник не дает игроку пройти мост, пока тот не вернется в деревню и не получит разрешение перейти мост у деревенского старосты. Стражник возвращается в поселение, оставляя мост незащищенным и доступным. Если игрок вернется в поселение и поговорит со стражником, тот отдаст ему легкие доспехи и щит.

Вернувшись назад, игрок продолжает свой путь, пересекает мост и попадает в башню, где встречает несколько демонов. Главный в этой части демон приказывает одному из других демонов бежать и сообщить их властелину (Злому Лорду) о вторжении игрока. Властелин не слишком обрадован этими новостями и, очевидно, убивает посыльного. Всякий раз, когда игрок входит в эту область, на него нападают монстры.

Игрок поднимается по скату и через вторую дверь уровня попадает на балкон башни, где его ждет обманчиво неподвижное каменное существо (Гранит). После разговора это создание атакует игрока. После смерти этого существа открывается путь в покои Злого Лорда.

Войдя в следующие покои игрок находит корень всех проблем жителей деревни — Злого Лорда. Произнеся несколько угроз Злой Лорд атакует игрока. Это финальная битва, и как только Злой Лорд будет уничтожен, игра заканчивается.

Чтобы создать представленный выше ход игры, вы должны тщательно спроектировать скрипты, которые получают управление каждый раз когда игрок говорит с конкретным персонажем, входит в определенную область карты или переходит на уровень. (Аспекты запуска скриптов были описаны в предыдущем разделе «Разработка скриптов».)

Большинство скриптов просты для понимания. Это триггеры карты, которые транспортируют игрока на другую карту каждый раз, когда игрок пытается выйти с текущей карты. Щелчок по персонажу запускает другой скрипт. Наиболее изобретательные применения скриптов — те, в которых проверяется была ли достигнута заданная точка маршрута.

Например, в начале игры исполняется скрипт `scene1.mls`. Тип персонажа игрока меняется на NPC и ему назначается маршрут. Это

заставляет игрока идти в деревню, и, как только он достигает конечной точки назначенного маршрута, в действие вступает новый скрипт, добавляющий к уровню монстра. Затем монстр следует по маршруту. Когда монстр достигает конечной точки, запускается новый скрипт, отображающий диалог между игроком и монстром. По завершении диалога тип персонажа игрока меняется обратно на РС, и начинается сражение.

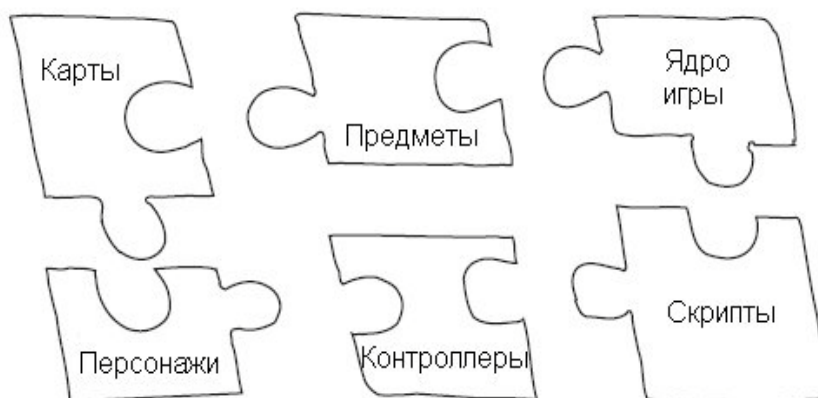
После боя в деревне игрок идет к центру поселения, что приводит к вызову скрипта телепортирующего игрока обратно в поселение — на этот раз срабатывает другая часть скрипта `scene1.mls`, добавляющая к сцене деревни ее жителей. Тот же самый стиль использования скриптов, где обработка скриптов запускается на основании маршрутов перемещения персонажей, используется и на нижнем уровне башни, когда демон убегает чтобы сообщить своему владыке о вторжении игрока.

Оставшиеся скрипты случайным образом добавляют монстров на карты, иногда основываясь на состоянии конкретных флагов скрипта. Например, если игрок убьет Гранита, устанавливается флаг, в дальнейшем сообщающий скрипту о том, что не надо добавлять Гранита на карту, когда персонаж входит на уровень балкона башни.

Использование флагов замечательно демонстрируется в *The Tower* — просмотрите каждый используемый файл скрипта, чтобы лучше понять применение флагов. Все игровые скрипты вы найдете на прилагаемом к книге CD-ROM.

## Программирование примера игры

Проект игры относительно прост. Главная работа — собрать все части, чтобы они работали сообща. Представьте игру, разделенную на основные компоненты, как показано на рис. 16.7. Если вы читаете книгу последовательно, то в предыдущих главах уже узнали как иметь дело с компонентами вашей игры. Теперь вам надо только собрать эти компоненты в пригодную к употреблению форму.



**Рис. 16.7.** Проект вашей игры разделен на части, подобно паззлу. У каждой части есть свое предназначение, и все части должны быть расположены на предназначенных им местах для создания целой игры

В этой книге я отделял каждый игровой компонент и не перемешивал их между главами (за исключением персонажей, которые зависят от нескольких компонентов из разных глав). Я не хотел, чтобы вы зависели от совместной работы каждого компонента с каждым. В этом случае вы можете выбрать те компоненты, которые точно соответствуют потребностям вашего проекта, и быстро и легко применить то, что узнали.

И, наконец, в этой главе вы откроете как легко взять все отдельные компоненты и разместить их вместе для образования законченной игры. Вооружившись проектом игры, набросанным в предыдущем разделе, «Проектирование примера игры», вы можете сосредоточиться на программной стороне создания примера игры.

В таблице 16.4 описаны компоненты, используемые в The Tower, и указаны главы, в которых эти компоненты были разработаны.

**Таблица 16.4.** Компоненты игры The Tower

<b>Компонент</b>	<b>Описание</b>
Ядро игры	Используется каждый компонент ядра игры, за исключением сетевого ядра. Конкретнее, это следующие компоненты: графическое ядро, системное ядро и ядро ввода. Целиком ядро игры рассматривалось в главе 6, «Создаем ядро игры»
Пирамида видимого пространства и текстовые окна	Компонент пирамиды видимого пространства из главы 8, «Создание трехмерного графического движка», используется для отбрасывания невидимых объектов до того, как они будут визуализированы. Кроме этого, класс текстового окна из главы 12 применяется для отображения диалогов и других текстов в игре.
Движок смешанной 2D/3D графики	Это тот же самый графический движок, который был разработан в главе 9. Он позволяет визуализировать трехмерные сетки поверх заранее визуализированных двухмерных фоновых изображений.
Скрипты и контроллер скриптов	Для разработки игровых скриптов используется система Mad Lib Script, созданная в главе 10. Класс контроллера скриптов из главы 12 применяется для загрузки и обработки этих скриптов в игре.
Предметы и список имущества	Главный список предметов в комбинации с системой управления имуществом персонажа, оба из главы 11, «Определение и использование объектов».
Персонажи и контроллер персонажей	Для управления персонажами и их визуализации в игре используется полный контроллер персонажей (с поддержкой главного списка персонажей), показанный в главе 12.
Заклинания и контроллер заклинаний	Контроллер заклинаний из главы 12 применяется для управления заклинаниями и их отображения. Главный список заклинаний (также описанный в главе 12) используется для определения заклинаний в игре.
Барьеры и триггеры	Барьеры блокируют перемещение, а триггеры исполняют скрипты, когда их кто-то коснется; обе системы обсуждались в главе 13, «Работа с картами и уровнями».

Проект с примером игры (\BookCode\Chap16\The Tower\)) состоит из перечисленных ниже файлов, представляющих игровые компоненты из таблицы 16.4.

- **Core\_Graphics.h, Core\_Graphics.cpp, Core\_Input.h, Core\_Input.cpp, Core\_Sound.h, Core\_Sound.cpp, Core\_System.h, Core\_System.cpp** и **Core\_Global.h**. Эти файлы составляют ядро игры.
- **Frustum.h, Frustum.cpp, Window.h** и **Window.cpp**. Эти файлы представляют объекты пирамиды видимого пространства и текстовых окон.
- **Script.h, Script.cpp, Game\_Script.h** и **Game\_Script.cpp**. В этих файлах находятся система MLS и производный класс обработчика.
- **Charics.h** и **Charics.cpp**. В этой паре файлов располагается система управления имуществом персонажей.
- **Chars.h, Chars.cpp, Game\_Chars.h** и **Game\_Chars.cpp**. Эти четыре файла хранят контроллер персонажей и код производного класса контроллера персонажей для примера игры.
- **Spell.h, Spell.cpp, Game\_Spell.h** и **Game\_Spell.cpp**. Эти четыре файла содержат контроллер заклинаний и производный класс контроллера заклинаний.
- **Barrier.h, Barrier.cpp, Trigger.h** и **Trigger.cpp**. Код из этих файлов обрабатывает барьеры и триггеры.
- **MCL.h, MIL.h** и **MSL.h**. Это заголовочные файлы главного списка персонажей, главного списка предметов и главного списка заклинаний.
- **WinMain.h** и **WinMain.cpp**. Это основные файлы приложения, содержащие код класса приложения.

Не пугайтесь количества используемых здесь файлов. Вы уже познакомились с большей их частью по ходу чтения книги. Однако для лучшего понимания как эти файлы работают вместе, возьмем их один за другим, начиная с главного класса приложения.

## Структурирование приложения

Главное приложение относительно небольшое (если вы можете назвать небольшим примерно 1500 строк кода). Его работа — инициализация всех необходимых компонентов и отслеживание состояния игры (верно, здесь используется обработка на основе состояний).

Сперва вы объявляете класс приложения. Хотя на данный момент класс незавершен, в оставшейся части главы все части займут свои места и класс приложения станет законченным. Сейчас посмотрим на разделы класса приложения где устанавливаются данные класса и инициализируется игровая система:

```
class cApp : public cApplication
{
    friend class cSpells;
    friend class cChars;
    friend class cGameScript;
```

---

**СОВЕТ**

Когда один класс объявляет другой как друга класса, как сделано здесь, этот дружественный класс получает неограниченный доступ к данным объявляющего класса.

---

Класс приложения начинается с установки ссылок для трех друзей класса. Эти три класса, **cSpells**, **cChars** и **cGameScript**, являются производными классами контроллеров для заклинаний, персонажей и скриптов, соответственно. Каждому из этих классов необходим особый доступ к классу приложения, поэтому мы делаем их друзьями. В следующей части класса **cApp** объявляется список относящихся к ядру игры объектов, которые все являются закрытыми членами класса **cApp**:

```
private:
    // Графическое устройство, камера и шрифт
    cGraphics m_Graphics;
    cCamera   m_Camera;
    cFont      m_Font;

    // Система ввода и устройства
    cInput      m_Input;
    cInputDevice m_Keyboard;
    cInputDevice m_Mouse;

    // Звуковая система, звуковые и музыкальные каналы
    // и звуковые данные
    cSound      m_Sound;
    cSoundChannel m_SoundChannel;
    cMusicChannel m_MusicChannel;
    cSoundData   m_SoundData;
```

Как видите, из графического ядра используются объекты графического устройства, шрифта и камеры. Для ввода здесь есть объект системы ввода, плюс устройства для клавиатуры и мыши. Завершает все набор объектов для использования звуковой системы, отдельные звуковой и музыкальный канал и объект звуковых данных для загрузки звуков.

Небольшое растровое изображение хранит графику, используемую для отображения полосы зарядки игрока (количество заряда, растущее для нападения). Это растровое изображение вы храните используя объект текстуры. Вслед за этим вы включаете три объекта текстовых окон для отображения различных диалогов и текстов на экране:

```
// Текстура с растровым изображением
cTexture m_Options;

// Текстовые окна
cWindow m_Stats;
cWindow m_Window;
cWindow m_Header;
```

В этой точке объявления класса вы определяете пару вспомогательных закрытых функций:

```
BOOL WinGame(); // Обработка сценария завершения игры

// Получение персонажа, на который указывает мышь
sCharacter *GetCharacterAt(long XPos, long YPos);
```

Функция **WinGame** вызывается всякий раз, когда скрипт сталкивается с действием завершения игры. Это действие запускает окончание игры, которое возвращает управление к главному меню. **GetCharacterAt** — это функция (обсуждавшаяся в главе 14, «Создание боевых последовательностей»), определяющая по какому из персонажей игрок щелкнул мышью.

Завершают **cApp** конструктор класса и переопределенные функции **Init**, **Shutdown** и **Frame**, все объявленные открытыми:

```
public:
    cApp();

    // Переопределенные функции
    BOOL Init();
    BOOL Shutdown();
    BOOL Frame();
};
```

Пока закрытые переменные не представляют собой ничего без поддерживающих функций. Сейчас вам следует сосредоточиться на четырех открытых функциях — конструкторе класса **cApp**, **Init**, **Shutdown** и **Frame**.

## Конструктор cApp

Конструктор класса приложения обычно используется для определения класса окна приложения, стиля, размера, имени класса и имени приложения, и прочих подобных вещей. Конструктор устанавливает только ширину и высоту окна, стиль окна, имя класса и заголовок приложения:

```
cApp::cApp()
{
    m_Width = 640;
    m_Height = 480;
    m_Style = WS_BORDER | WS_CAPTION |
              WS_MINIMIZEBOX | WS_SYSMENU;
    strcpy(m_Class, "GameClass");
    strcpy(m_Caption, "The Tower by Jim Adams");
}
```

## Функция приложения Init

Начальная точка игры, функция **Init**, инициализирует систему (включая графическую, звуковую системы и систему ввода), устанавливает контроллеры персонажей и заклинаний, загружает главный список



предметов, заталкивает в стек состояние главного меню и выполняет еще несколько разнообразных функций. Рассмотрим функцию **Init** фрагмент за фрагментом, чтобы увидеть, что в ней происходит:

```
BOOL cApp::Init()
{
    // Инициализация графического устройства
    m_Graphics.Init();

    // Определяем, используется ли полноэкранный режим
#ifdef FULLSCREENMODE
    m_Graphics.SetMode(GethWnd(), FALSE, TRUE, 640, 480, 16);
#else
    m_Graphics.SetMode(GethWnd(), TRUE, TRUE);
#endif

    // Установка перспективы
    m_Graphics.SetPerspective(0.6021124f, 1.33333f, 1.0f, 20000.0f);

    // Разрешаем отображение курсора
    ShowMouse(TRUE);

    // Создаем шрифт
    m_Font.Create(&m_Graphics, "Arial", 16, TRUE);
}
```

Здесь главное графика — вы инициализируете графическую систему и устанавливаете видеорежим. Макроопределение в начале файла `WinMain.cpp` определяет, будет ли использоваться полноэкранный режим — просто закомментируйте эту строчку, если хотите работать в оконном режиме. Затем вы устанавливаете перспективу, чтобы она соответствовала программе трехмерного моделирования в которой визуализировались задники. В конце вы создаете шрифт для использования в игре и включаете отображение указателя мыши.

Далее вы инициализируете систему ввода и создаете два интерфейса устройств — один для клавиатуры и другой для мыши:

```
// Инициализация системы и устройств ввода
m_Input.Init(GethWnd(), GethInst());
m_Keyboard.Create(&m_Input, KEYBOARD);
m_Mouse.Create(&m_Input, MOUSE, TRUE);
```

Завершив код инициализации графического ядра, вы инициализируете звуковую систему и создаете звуковой и музыкальный каналы:

```
// Инициализация звуковой системы и каналов
m_Sound.Init(GethWnd(), 22050, 1, 16);
m_SoundChannel.Create(&m_Sound, 22050, 1, 16);
m_MusicChannel.Create(&m_Sound);
```

Теперь вы инициализируете относящиеся к игре данные и интерфейсы. Вы загружаете главный список предметов (находящийся в каталоге `\BookCode\Chap16\Data`) и инициализируете контроллер персонажей и контроллер заклинаний:

```

// Загрузка главного списка предметов
FILE *fp;

// Обнуляем память, используемую для хранения данных предметов
for(long i = 0; i < 1024; i++)
    ZeroMemory(&m_MIL[i], sizeof(sItem));

if((fp = fopen("../Data\\Game.mil", "rb")) != NULL) {
    for(i = 0; i < 1024; i++)
        fread(&m_MIL[i], 1, sizeof(sItem), fp);
    fclose(fp);
}

// Инициализация контроллера персонажей
m_CharController.SetData(this);
m_CharController.Init(&m_Graphics, &m_Font,
    "../Data\\Game.mcl", (sItem*)&m_MIL,
    m_SpellController.GetSpell(0),
    sizeof(g_CharMeshNames)/sizeof(char*), g_CharMeshNames,
    "../Data\\", "../Data\\",
    sizeof(g_CharAnimations) / sizeof(sCharAnimationInfo),
    (sCharAnimationInfo*)&g_CharAnimations,
    &m_SpellController);

// Инициализация контроллера заклинаний
m_SpellController.SetData(this);
m_SpellController.Init(&m_Graphics,
    "../Data\\Game.msl",
    sizeof(g_SpellMeshNames)/sizeof(char*), g_SpellMeshNames,
    "../Data\\", &m_CharController);

```

В вызовах инициализации контроллеров много аргументов (обратитесь к главе 12, чтобы увидеть, что делает каждый из аргументов). Вы находитесь на половине пути через функцию **Init**. В этом месте вы загружаете растровое изображение, содержащее графику для отображения заряда игрока, а также создаете и позиционируете текстовые окна:

```

// Получаем растровое изображение параметров
m_Options.Load(&m_Graphics, "../Data\\Options.bmp");

// Создаем главное окно, заголовок и окно статистики
m_Window.Create(&m_Graphics, &m_Font);
m_Header.Create(&m_Graphics, &m_Font);
m_Stats.Create(&m_Graphics, &m_Font);

// Позиционируем все окна
m_Window.Move(2,2, 636, 476);
m_Header.Move(2,2,128,32,-1,-1,D3DCOLOR_RGBA(128,16,16,255));
m_Stats.Move(2,2,128,48);

```

Завершая функцию **Init** вы вызываете производный класс скрипта, чтобы сообщить скрипту интерфейс какого класса приложения использовать. Вслед за этим вы помещаете в стек состояний первое игровое состояние, главное меню:

```

// Устанавливаем указатель на приложение для скрипта
m_Script.SetData(this);

// Помещаем в стек состояние главного меню,
// предварительно установив параметры меню

```

```
g_MenuOptions = MENU_LOAD;
m_StateManager.Push(MenuFrame, this);

return TRUE;
}
```

## Функция *Shutdown*

Что хорошего в функции **Init** без соответствующей функции **Shutdown** для отключения и освобождения используемых в игре ресурсов? Функция **cApp::Shutdown** делает следующее: очищает контроллеры, удаляет состояния из стека состояний, освобождает скрипты и так далее:

```
BOOL cApp::Shutdown()
{
    // Извлекаем все состояния
    m_StateManager.PopAll(this);

    // Освобождаем контроллеры
    m_CharController.Free();
    m_SpellController.Free();

    // Освобождаем объект скриптов
    m_Script.Free();

    // Освобождаем данные уровня
    FreeLevel();

    // Освобождаем текстуру параметров
    m_Options.Free();

    // Освобождаем текстовые окна
    m_Window.Free();
    m_Header.Free();
    m_Stats.Free();

    // Отключаем звук
    m_MusicChannel.Free();
    m_SoundChannel.Free();
    m_Sound.Shutdown();

    // Отключаем ввод
    m_Keyboard.Free();
    m_Mouse.Free();
    m_Input.Shutdown();

    // Отключаем графику
    m_Font.Free();
    m_Graphics.Shutdown();

    return TRUE;
}
```

## Обработка кадров в функции *Frame*

Для каждого кадра обновления игры вызывается функция класса приложения **Frame**. Однако, чтобы ограничить насколько часто игра будет действительно обновляться, поддерживается таймер, ограничивающий

обработку кадров 30 кадрами в секунду. Процесс ограничения обновлений занимает первую половину функции **Frame** и показан ниже:

```

BOOL cApp::Frame()
{
    static DWORD UpdateTimer = timeGetTime();

    // Ограничиваем обновление кадров до 30 fps
    if(timeGetTime() < UpdateTimer + 33)
        return TRUE;
    UpdateTimer = timeGetTime();
}

```

Как я упоминал, игра обновляется 30 раз в секунду. В каждом кадре обновления игры считывается состояние клавиатуры и мыши и обрабатывается текущее игровое состояние:

```

// Захватываем устройства ввода и считываем с них
// данные для всех состояний
m_Keyboard.Acquire(TRUE); // Чтение клавиатуры
m_Keyboard.Read();
m_Mouse.Acquire(TRUE); // Чтение мыши
m_Mouse.Read();

// Обработка состояния, возвращение результата
return m_StateManager.Process(this);
}

```

Глава 1, «Подготовка к работе с книгой», рассказывает как работает обработка на основе состояний. Поскольку состояния помещаются в стек состояний, при вызове **cStateManager::Process**, который вы можете видеть в функции **Frame**, выполняется самое верхнее состояние.

## Использование обработки на основе состояний

Я разработал пример игры с использованием обработки на основании состояний, чтобы эффективно использовать структуру обработки класса приложения. Игра использует следующие четыре состояния:

- **Главное меню.** Когда выполняется это состояние игра отображает главное меню, давая игроку возможность выбрать начало новой игры, загрузку игры, возобновление или запись текущей игры, либо выход из игры.
- **Игра.** Это состояние используется наиболее часто, поскольку отвечает за обновление и визуализацию каждого кадра в игре.
- **Окно статистики персонажа.** Каждый раз, когда игрок в ходе игрового процесса щелкает правой кнопкой мыши, он получает доступ к окну статистики персонажа. Здесь игрок может использовать предметы, а также добавить предмет в экипировку или удалить из нее просто щелкнув по нему, кроме того, здесь можно проверить характеристики персонажа и известные заклинания.
- **Окно торговли.** Когда игрок разговаривает с жителями деревни, открывается окно торговли для покупки предметов. Щелкните по

предмету для его покупки или нажмите Esc, либо правую кнопку мыши для выхода.

Вы используете объект диспетчера состояний (смотрите главу 1) для управления обработкой этих четырех состояний. У каждого состояния есть связанная с ним функция, объявленная в классе **cApp**:

```
// Диспетчер обработки состояний и функции состояний
cStateManager m_StateManager;

static void MenuFrame(void *Ptr, long Purpose);
static void GameFrame(void *Ptr, long Purpose);
static void StatusFrame(void *Ptr, long Purpose);
static void BarterFrame(void *Ptr, long Purpose);
```

Четыре состояния управляют обработкой пользовательского ввода и визуализацией результатов на экране. Код каждой из функций состояний немного велик, чтобы быть приведенным здесь, поэтому вместо того, чтобы показывать код каждой функции, я сосредоточусь на наиболее важной функции состояния **GameFrame** (вызываемой в каждом кадре для обновления игрока и монстров и визуализации сцены):

```
void cApp::GameFrame(void *Ptr, long Purpose)
{
    cApp *App = (cApp*)Ptr;
    sCharacter *CharPtr;
    BOOL MonstersInLevel;
    long TriggerNum;
    char Filename[MAX_PATH], Stats[256];
    float MaxY;

    // Обработываем только кадры
    if(Purpose != FRAMEPURPOSE)
        return;
```

Поскольку это состояние кадра, вы можете вызвать функцию **GameFrame** для одной из трех целей — инициализации состояния, обработки кадра и для отключения состояния. Функция **GameFrame** используется только с целью обновления кадра, поэтому обработка сразу завершается, если указана любая другая цель вызова.

---

**ПРИМЕЧАНИЕ**

В главе 1 вы читали об использовании *целей вызова (calling purpose)*. При использовании функций состояния цель вызова информирует функцию, зачем она была вызвана: для инициализации данных, либо для обработки кадра, либо для выключения и освобождения ресурсов.

---

В начале функции **GameFrame** производится быстрая проверка, нажата ли клавиша **Esc**. Если да, то в стек помещается состояние главного меню:

```
// Выход в экран меню, если нажата ESC
if(App->m_Keyboard.GetKeyState(KEY_ESC) == TRUE) {
    // Установка параметров меню
    g_MenuOptions = MENU_BACK | MENU_SAVE | MENU_LOAD;
```

Чтобы знать, какие команды отображать в главном меню, вы объявляете в начале приложения глобальную переменную. Эта глобальная переменная, **g\_MenuOptions** является битовым полем и использует следующие макроопределения для задания значений — **MENU\_BACK** для отображения команды возврата к игре, **MENU\_SAVE** для отображения команды сохранения игры и **MENU\_LOAD** для отображения команды загрузки игры. Как только команды определены, состояние можно помещать в стек:

```
// Помещаем в стек состояние главного меню
App->m_StateManager.Push(App->MenuFrame, App);
return;
}

// Если телепортировались, выполняем начальную обработку
// и возвращаемся
if(App->m_TeleportMap != -1) {
    // Освобождаем уровень и начинаем работу с новым
    App->FreeLevel();
    App->LoadLevel(App->m_TeleportMap);
    App->m_TeleportMap = -1; // Очищаем номер карты телепортирования
    return;                // Больше нечего обрабатывать в этом кадре
}
```

Каждый раз, когда игроку надо переместиться с одной карты на другую (например, при вызове функции **SetupTeleport** с номером карты и координатами, в которые перемещается игрок), вы присваиваете глобальной переменной с именем **m\_TeleportMap** допустимый номер карты для телепортации. Показанный выше фрагмент кода в каждом кадре проверяет, была ли установлена эта переменная, и телепортирует игрока на соответствующую карту.

Мы добрались до основной части функции **GameFrame**. В начале следующего блока кода вы сбрасываете флаг, отслеживающий, есть ли на карте какие-либо монстры. Затем сканируется весь список загруженных персонажей. Если в списке персонажей вы находите монстра, флаг **MonstersInLevel** устанавливается. Также для каждого монстра на карте меняются параметры его искусственного интеллекта в зависимости от заряда. Для заряда меньше 70, поведение монстра устанавливается на хождение по карте (что позволяет монстру перемещаться по уровню). Если заряд больше 70, поведение монстра настраивается на преследование игрока (это означает, что монстр пытается атаковать игрока):

```
// Отмечаем отсутствие монстров на уровне
MonstersInLevel = FALSE;

// Смотрим, есть ли какие-либо персонажи на уровне.
// Если есть какие-либо монстры, отмечаем это и устанавливаем
// их поведение на ходьбу по уровню, если заряд меньше 70
// и на преследование в ином случае.
// Также обрабатываем достижение персонажем маршрутной точки.
CharPtr = App->m_CharController.GetParentCharacter();
while(CharPtr != NULL) {
    // Меняем поведение монстра в зависимости от заряда
    if(CharPtr->Type == CHAR_MONSTER) {
```

```
MonstersInLevel = TRUE;

// Меняем AI в зависимости от заряда
if(CharPtr->Charge >= 70.0f) {
    CharPtr->AI = CHAR_FOLLOW;
    CharPtr->TargetChar = g_PCChar;
    CharPtr->Distance = 0.0f;
} else {
    CharPtr->AI = CHAR_WANDER;
}
}
```

Если же, с другой стороны, на карте найден NPC и его искусственный интеллект настроен на следование по маршруту, вызывается отдельная функция, определяющая, достиг ли персонаж конечной точки назначенного маршрута. Если достигнута последняя точка маршрута, выполняется скрипт завершения маршрута этого персонажа:

```
// Проверяем, достиг ли NPC последней точки маршрута
if(CharPtr->Type==CHAR_NPC && CharPtr->AI==CHAR_ROUTE) {
    // Была достигнута конечная точка?
    if(App->LastPointReached(CharPtr) == TRUE) {
        // Обрабатываем скрипт завершения маршрута персонажа
        sprintf(Filename, "..\\Data\\EOR%lu.mls", CharPtr->ID);
        App->m_Script.Execute(Filename);

        // Больше нечего обрабатывать в этом кадре
        return;
    }
}

// Переходим к следующему персонажу
CharPtr = CharPtr->Next;
}
```

Пройдя фазу сканирования персонажей в функции **GameFrame**, вы сравниваете флаг **MonstersInLevel** тем же самым флагом, сохраненным в предыдущем кадре. Если они не совпадают, значит началось или закончилось сражение, и вызывается соответствующий скрипт:

```
// Обработка начала сражения
if(MonstersInLevel == TRUE && App->m_MonstersLastFrame == FALSE)
    App->StartOfCombat();

// Обработка конца сражения, если оно завершилось
if(MonstersInLevel == FALSE && App->m_MonstersLastFrame == TRUE)
    App->EndOfCombat();

// Запоминаем, были ли монстры в этом кадре
// и восстанавливаем полный заряд игрока, если монстров нет
if((App->m_MonstersLastFrame == MonstersInLevel) == FALSE)
    g_PCChar->Charge = 100.0f;

// Обновление контроллеров
App->m_CharController.Update(33);
App->m_SpellController.Update(33);
```

Затем наступает момент, когда следует обновить все персонажи и заклинания. Поскольку функция **cApp::Frame** ограничивает обновление

игры 30 кадрами в секунду, все контроллеры используют время обновления 33 миллисекунды. Заметьте, что перед началом обновления измеритель заряда игрока восстанавливается до максимума, если на текущей карте нет монстров.

После обновления персонажей в игру вступают объекты триггеров. Если игрок входит в активный триггер, выполняется соответствующий скрипт:

```
// Проверка триггеров и исполнение скрипта
if((TriggerNum = App->m_Trigger.GetTrigger(g_PCChar->XPos,
                                           g_PCChar->YPos,
                                           g_PCChar->ZPos))) {
    sprintf(Filename, "..\\Data\\Trig%lu.mls", TriggerNum);
    App->m_Script.Execute(Filename);

    return; // Больше нечего обрабатывать в этом кадре
}
```

Сейчас вы будете визуализировать сцену, вызвав функцию **RenderFrame** из класса приложения. Функция **RenderFrame** визуализирует только задний фон и персонажей на карте — остальная часть кода рассматриваемой нами функции рисует окно статуса и измеритель заряда:

```
// Местоположение камеры в сцене
App->m_Graphics.SetCamera(&App->m_Camera);

// Визуализируем все
App->m_Graphics.ClearZBuffer();
if(App->m_Graphics.BeginScene() == TRUE) {
    App->RenderFrame(33);
}
```

Итак, вы сперва визуализируете сцену, используя функцию **RenderFrame**, которая получает в качестве аргумента количество времени (в миллисекундах), на которое должна быть обновлена анимация. Затем вы рисуете измеритель заряда игрока, но только если на карте есть монстры (это определяется по флагу **MonstersInLevel**):

```
// Визуализируем полосу заряда игрока,
// но только во время битвы
if(MonstersInLevel == TRUE) {
    D3DXMATRIX matWorld, matView, matProj;
    D3DVIEWPORT9 vpScreen;
    D3DXVECTOR3 vecPos;

    // Получаем мировое преобразование, преобразование вида
    // и преобразование проекции
    D3DXMatrixIdentity(&matWorld);
    App->m_Graphics.GetDeviceCOM()->GetTransform(
        D3DTS_VIEW, &matView);
    App->m_Graphics.GetDeviceCOM()->GetTransform(
        D3DTS_PROJECTION, &matProj);

    // Получаем порт просмотра
    App->m_Graphics.GetDeviceCOM()->GetViewport(&vpScreen);

    // Смещаем полосу заряда на высоту персонажа
```



```
g_PCChar->Object.GetBounds(NULL, NULL, NULL,
                             NULL, &MaxY, NULL, NULL);

// Проецируем координаты на экран
D3DXVec3Project(&vecPos, &D3DXVECTOR3(
    g_PCChar->XPos,
    g_PCChar->YPos + MaxY,
    g_PCChar->ZPos),
    &vpScreen, &matProj, &matView, &matWorld);

// Перемещаем на 8 пикселей вправо перед отображением
vecPos.x += 8.0f;
```

Здесь работают многочисленные функции, относящиеся к матрицам и векторам — вы используете их для вычисления экранных координат, в которых следует рисовать измеритель заряда. Чтобы определить, где рисовать измеритель, используется показанная выше функция **D3DXVec3Project**, вычисляющая двухмерные координаты на основании трехмерных мировых координат игрока.

Теперь вы отключаете Z-буфер и рисуете измеритель заряда, используя в качестве источника его изображения объект текстуры **m\_Option**:

```
// Отображаем полосу заряда рядом с игроком
// (мерцает, если полная)
App->m_Graphics.EnableZBuffer(FALSE);
App->m_Graphics.BeginSprite();
App->m_Options.Blit((long)vecPos.x, (long)vecPos.y,
    0, 0, 16, 4);
if(g_PCChar->Charge >= 100.0f) {
    if(timeGetTime() & 1)
        App->m_Options.Blit((long)vecPos.x, (long)vecPos.y,
            0, 4, 16, 4);
} else {
    App->m_Options.Blit((long)vecPos.x, (long)vecPos.y,
        0, 4, (long)(g_PCChar->Charge/100.0f*16.0f), 4);
}
App->m_Graphics.EndSprite();
}
```

В процессе игры вы отображаете статистику игрока в левом верхнем углу экрана — в нее включаются очки здоровья и маны, их текущий и максимальный уровень:

```
// Рисуем статистику игрока вверху слева
sprintf(Stats, "%ld / %ld HP\r\n%ld / %ld MP",
    g_PCChar->HealthPoints, g_PCChar->Def.HealthPoints,
    g_PCChar->ManaPoints, g_PCChar->Def.ManaPoints);
App->m_Stats.Render(Stats);
App->m_Graphics.EndScene();
}

App->m_Graphics.Display();
}
```

Функция **GameFrame** завершается вызовом **EndScene** и отображением кадра пользователю. Остальные функции состояний по своей

природе просты, поэтому я только кратко опишу их здесь, оставив их исследование вам, когда вы будете работать с примером игры на CD-ROM.

Вы используете функцию **MenuFrame** для отображения главного меню, которое, во всей своей красе, представляет собой вращающийся текстурированный полигон, поверх которого располагаются команды главного меню. Целью функции **MenuFrame** является отслеживание того, какая из команд главного меню выбрана, и вызов соответствующих функций.

Функцию **StatusFrame** вы используете для показа статистики игрока (очков здоровья, очков маны, известных заклинаний и т.д.) когда отображается окно статистики игрока. Эта функция обрабатывает экипировку предметами и проверку статистики игрока. Последняя из функций состояния, **BarterFrame**, показывает склад лавочника и позволяет игроку щелчком выбрать предметы для покупки.

## Работа с картами

Пример игры разделен на пять карт (сцен). Каждая сцена использует шесть растровых изображений, которые загружаются как текстуры и рисуются на экране в каждом кадре. Кроме того игра использует для каждой сцены скрытую упрощенную сетку. Эти упрощенные сетки (как описывалось в главе 9) помогают правильно рисовать трехмерных персонажей, населяющих каждую сцену.

Для загрузки и использования шести текстур объявите массив объектов **cTexture** (для хранения шести растровых изображений), объект **cMesh** (который содержит упрощенную сетку сцены) и объект **cObject** (используемый для визуализации упрощенной сетки):

```
long      m_SceneNum;           // Номер текущей сцены, 1-5
cTexture  m_SceneTextures[6];   // Шесть текстур сцены
cMesh     m_SceneMesh;          // Упрощенная сетка сцены
cObject   m_SceneObject;        // Объект упрощенной сцены
```

Для работы со сценами используются четыре функции класса приложения. Это функции **LoadLevel**, **FreeLevel**, **GetHeightBelow** и **CheckIntersect**. Вы используете функции **GetHeightBelow** и **CheckIntersect**, представленные в главе 8, для проверки пересечения сетки с сеткой. В данном случае проверка пересечения сеток применяется для обнаружения тех случаев, когда персонаж пересекается с упрощенной сеткой сцены.

Функция **LoadLevel** загружает шесть текстур сцены и упрощенную сетку и исполняет скрипт, связанный с загрузкой сцены. Внешний файл, который вы скоро увидите, хранит местоположение камеры в каждой сцене. Вот код **LoadLevel**:

```
BOOL cApp::LoadLevel(long Num)
{
    char Filename[MAX_PATH];
    FILE *fp;
```

```
long i;
float XPos, YPos, ZPos, XAt, YAt, ZAt;

FreeLevel(); // Освобождаем предыдущий уровень

// Сохраняем номер сцены
m_SceneNum = Num;
```

Сейчас ранее загруженный уровень освобожден и сохранен номер новой сцены. Далее вы загружаете текстуры сцены и упрощенную сетку:

```
// Загрузка фоновых текстур
for(i = 0; i < 6; i++) {
    sprintf(Filename, "..\\Data\\Scene%u%u.bmp", Num, i+1);
    if(m_SceneTextures[i].Load(&m_Graphics, Filename) == FALSE)
        return FALSE;
}

// Загрузка сетки сцены и конфигурирование объекта
sprintf(Filename, "..\\Data\\Scene%u.x", Num);
if(m_SceneMesh.Load(&m_Graphics, Filename) == FALSE)
    return FALSE;
m_SceneObject.Create(&m_Graphics, &m_SceneMesh);
```

Загрузив сетку сцены и создав объект сцены вы готовы определить местоположение камеры, используемой для визуализации трехмерной графики. Для размещения камеры в каждой из сцен вы заранее создаете текстовые файлы для каждой сцены. Имена этих файлов `cam1.txt`, `cam2.txt`, `cam3.txt`, `cam4.txt` и `cam5.txt` — каждое имя сформировано согласно номеру соответствующей сцены (сцены нумеруются от 1 до 5).

Размещение камеры в сцене заключается в открытии соответствующего текстового файла и чтении шести чисел, каждое из которых используется для определения ориентации камеры в сцене. Первые три числа представляют местоположение камеры в мире, а последние три числа это координаты точки, на которую направлена камера.

После того, как вы прочитали шесть чисел и сориентировали камеру, вызовите функцию **`cGraphics::SetCamera`**, чтобы проинформировать Direct3D о новом преобразовании вида, которое будет использоваться камерой:

```
// Загрузка данных камеры
sprintf(Filename, "..\\Data\\Cam%u.txt", Num);
if((fp=fopen(Filename, "rb")) == NULL)
    return FALSE;

XPos = GetNextFloat(fp);
YPos = GetNextFloat(fp);
ZPos = GetNextFloat(fp);
XAt  = GetNextFloat(fp);
YAt  = GetNextFloat(fp);
ZAt  = GetNextFloat(fp);

fclose(fp);

// Позиционирование камеры для сцены
```

```
m_Camera.Point(XPos, YPos, ZPos, XAt, YAt, ZAt);
m_Graphics.SetCamera(&m_Camera);
```

После того, как вы спозиционировали камеру согласно файлу, класс очищает флаг, определяющий наличие монстров в текущей сцене (для обработки сражений), и затем выполняет связанный со сценой скрипт:

```
// Нет монстров в последнем кадре
m_MonstersLastFrame = FALSE;

// Выполняем скрипт загрузки сцены
sprintf(Filename, "..\\Data\\Scene%lu.mls", Num);
m_Script.Execute(Filename);

return TRUE;
}
```

Как видите, **LoadLevel** делает не так уж много. Функция **FreeLevel** доставит еще меньше беспокойства. Она освобождает текстуры сцены и упрощенную сетку, удаляет всех персонажей из контроллера персонажей (естественно, исключая игрока) и очищает все обрабатываемые в данный момент заклинания. Вот полный код функции **FreeLevel** (без комментариев, поскольку функция прямолинейна):

```
BOOL cApp::FreeLevel()
{
    sCharacter *CharPtr, *NextChar;
    long i;

    // Освобождение сетки сцены и текстур
    m_SceneMesh.Free();
    m_SceneObject.Free();
    for(i = 0; i < 6; i++)
        m_SceneTextures[i].Free();

    // Освобождение триггеров и барьеров
    m_Barrier.Free();
    m_Trigger.Free();

    // Освобождение всех независимых персонажей
    if((CharPtr = m_CharController.GetParentCharacter()) != NULL) {
        while(CharPtr != NULL) {
            // Запоминаем следующий персонаж
            NextChar = CharPtr->Next;

            // Удаляем независимый персонаж
            if(CharPtr->Type != CHAR_PC)
                m_CharController.Remove(CharPtr);

            // Переходим к следующему персонажу
            CharPtr = NextChar;
        }
    }

    // Освобождаем все эффекты заклинаний
    m_SpellController.Free();

    return TRUE;
}
```

## Использование барьеров и триггеров

В игре The Tower используются и барьеры и триггеры. Эти компоненты остались практически неизменными по сравнению с показанными в главе 13, так что можете обратиться к этой главе за более подробными сведениями об их использовании. Только скрипты имеют возможность добавлять барьеры и триггеры в игру. Вы объявляете объекты барьеров и триггеров в объявлении класса **cApp** следующим образом:

```
cTrigger m_Trigger;  
cBarrier m_Barrier;
```

Повторюсь, в этих интерфейсах ничего не изменилось, так что вы сейчас можете сосредоточиться на том, как осуществляется управление персонажами в игре.

## Управление персонажами

Персонажи — это сердце и душа вашей игры. Контроллеры персонажей и заклинаний, разработанные в главе 12, замечательно подходят для примера игры из этой главы. Если вы читали главу 12, то, вероятно, вспомните, что контроллеры должны наследоваться, так что давайте с этого и начнем наши дела здесь.

Вы наследуете контроллер персонажей для управления игроком в игре и проверки столкновений перемещающихся персонажей с картой. В игре The Tower вы используете производный контроллер персонажей, прототип которого был показан в главе 12, для управления всеми игровыми персонажами. Первый шаг для использования контроллера персонажей в игре — это наследование вашего собственного класса от **cCharacterController**:

---

<b>ПРИМЕЧАНИЕ</b>	Производный класс контроллера персонажей находится в паре файлов с именами <code>Game_Chars.h</code> и <code>Game_Chars.cpp</code> .
-------------------	--

---

```
class cChars : public cCharacterController  
{  
private:  
    cApp *m_App;  
  
    BOOL PCUpdate(sCharacter *Character, long Elapsed,  
                  float *XMove, float *YMove, float *ZMove);  
    BOOL ValidateMove(sCharacter *Character,  
                     float *XMove, float *YMove, float *ZMove);  
  
    BOOL Experience(sCharacter *Character, long Amount);  
    BOOL PCTeleport(sCharacter *Character, sSpell *Spell);  
  
    BOOL ActionSound(sCharacter *Character);  
  
    BOOL DropMoney(float XPos, float YPos, float ZPos,  
                   long Quantity);  
    BOOL DropItem(float XPos, float YPos, float ZPos,  
                  long Item, long Quantity);  
};
```

```
public:
    BOOL SetData(cApp *App) { m_App = App; return TRUE; }
};
```

Класс **cChars** предлагает только одну открытую функцию, **SetData**. Вы используете функцию **SetData** для установки указателя на класс приложения в экземпляре класса **cChars**. Кроме того, класс **cChars** переопределяет только функции, используемые для перемещения игрока и проверки допустимости перемещений всех персонажей. Остальные функции вступают в игру когда игрок получает очки опыта после сражения или телепортирует персонаж заклинанием, когда контроллер персонажей воспроизводит звук, когда монстр роняет немного денег или когда монстр роняет предмет после своей смерти.

Поскольку производному классу контроллера персонажа требуется доступ к классу приложения, вы должны вызвать функцию **SetData** перед вызовом любых других функций класса **cChar**. Функция **SetData** получает один аргумент — указатель на класс приложения.

Прочие функции, такие как **Experience**, **DropMoney** и **DropItem** сообщают игровому движку что монстр был убит и игра должна вознаградить игрока очками опыта, а также деньгами и предметами, выроненными умирающим монстром. Эти награды копятся до конца сражения, и тогда их обрабатывает функция класса приложения **EndOfCombat**.

Нас здесь в основном интересует функция **PCUpdate**. Она определяет, какие клавиши и кнопки мыши нажал игрок. Сейчас мы возьмем эту функцию, чтобы препарировать ее:

```
BOOL cChars::PCUpdate(sCharacter *Character, long Elapsed,
                    float *XMove, float *YMove, float *ZMove)
{
    float Speed;
    sCharacter *TargetChar;
    float XDiff, YDiff, ZDiff;
    float Dist, Range;
    char Filename[MAX_PATH];
    long Spell = -1;
```

Функция **PCUpdate** начинается с прототипа и объявления нескольких переменных. Функция **PCUpdate** использует пять аргументов — указатель на обновляемый персонаж, время (в миллисекундах), прошедшее с момента последнего обновления, и три указателя на переменные, которые будут заполнены перемещением персонажа по каждой из осей.

Начинается **PCUpdate** с определения того, следует ли проводить обновление (основываясь на прошедшем времени) и продолжается определением того, какие клавиши были нажаты на клавиатуре (если это имело место). Если была нажата клавиша перемещения курсора вверх, персонаж перемещается вперед, а если были нажаты клавиши перемещения курсора влево или вправо, персонаж поворачивается, как показано ниже:

```
// Не обновляем, если не прошло времени
if(!Elapsed)
    return TRUE;

// Поворот персонажа
if(m_App->m_Keyboard.GetKeyState(KEY_LEFT) == TRUE) {
    Character->Direction -= (float)Elapsed / 1000.0f * 4.0f;
    Character->Action = CHAR_MOVE;
}

if(m_App->m_Keyboard.GetKeyState(KEY_RIGHT) == TRUE) {
    Character->Direction += (float)Elapsed / 1000.0f * 4.0f;
    Character->Action = CHAR_MOVE;
}

if(m_App->m_Keyboard.GetKeyState(KEY_UP) == TRUE) {
    Speed = (float)Elapsed / 1000.0f *
        m_App->m_CharController.GetSpeed(Character);
    *XMove = (float)sin(Character->Direction) * Speed;
    *ZMove = (float)cos(Character->Direction) * Speed;
    Character->Action = CHAR_MOVE;
}
```

Для каждого производимого игроком перемещения, такого как ходьба вперед или поворот налево и направо, вам необходимо установить действие персонажа игрока в **CHAR\_MOVE**. Заметьте, что хотя нажатие клавиш перемещения курсора влево и вправо немедленно разворачивает персонаж игрока, код не модифицирует немедленно координаты персонажа. Вместо этого вы сохраняете направление перемещения в переменных **XMove** и **ZMove**.

Затем вы определяете, щелкнул ли игрок левой кнопкой мыши. Вспомните, что согласно проекту примера игры, щелчок левой кнопкой мыши по расположенному вблизи персонажу приводит либо к атаке персонажа (если этот персонаж монстр) или к разговору с персонажем (если персонаж NPC):

```
// Обработка действия атаки/разговора
if(m_App->m_Mouse.GetButtonState(MOUSE_LBUTTON) == TRUE) {
    // Смотрим, какой персонаж выбран и проверяем,
    // является ли он монстром
    if((TargetChar = m_App->GetCharacterAt(
        m_App->m_Mouse.GetXPos(),
        m_App->m_Mouse.GetYPos())) != NULL) {
```

Показанный выше фрагмент кода просто вызывает функцию **GetCharacterAt**, сканирующую персонажей, расположенных под указателем мыши. Если персонаж найден, вы определяете его тип; для NPC вы выполняете соответствующий скрипт персонажа:

```
// Обработка разговора с NPC
if(TargetChar->Type == CHAR_NPC) {
    // Не проверяем расстояние, просто обрабатываем скрипт
    sprintf(Filename, "..\\Data\\Char%lu.mls",
        TargetChar->ID);
    m_App->m_Script.Execute(Filename);
}
```

```

        return TRUE; // Больше нечего обрабатывать
    }

```

С другой стороны, если персонаж, по которому щелкнули, является монстром и находится на допустимом для атаки расстоянии, вы начинаете боевые действия:

```

// Обработка атаки монстра
if(TargetChar->Type == CHAR_MONSTER) {
    // Получаем расстояние до цели
    XDiff = (float)fabs(TargetChar->XPos - Character->XPos);
    YDiff = (float)fabs(TargetChar->YPos - Character->YPos);
    ZDiff = (float)fabs(TargetChar->ZPos - Character->ZPos);
    Dist = XDiff*XDiff + YDiff*YDiff + ZDiff*ZDiff;

    // Смещаем расстояние на радиус цели
    Range = GetXZRadius(TargetChar);
    Dist -= (Range * Range);

    // Получаем максимальную дальность атаки
    Range = GetXZRadius(Character);
    Range += Character->Def.Range;

    // Выполняем атаку только если цель в заданном диапазоне
    if(Dist <= (Range * Range)) {
        // Устанавливаем информацию цели/жертвы
        TargetChar->Attacker = Character;
        Character->Victim = TargetChar;

        // Поворачиваемся к жертве
        XDiff = TargetChar->XPos - Character->XPos;
        ZDiff = TargetChar->ZPos - Character->ZPos;
        Character->Direction = (float)atan2(XDiff, ZDiff);

        // Устанавливаем действие
        m_App->m_CharController.SetAction(Character,
                                           CHAR_ATTACK);
    }
}
}

```

Подходя к концу функции **PCUpdate**, контроллеру необходимо определить, какое заклинание произносится на близстоящего персонажа. В игре для произнесения заклинания надо навести на персонажа указатель мыши и нажать одну из цифровых клавиш (от 1 до 5):

```

// Произносим заклинание, основываясь на нажатой цифре
if(m_App->m_Keyboard.GetKeyState(KEY_1) == TRUE) {
    m_App->m_Keyboard.SetLock(KEY_1, TRUE);
    Spell = 0; // Огненный шар
}

if(m_App->m_Keyboard.GetKeyState(KEY_2) == TRUE) {
    m_App->m_Keyboard.SetLock(KEY_2, TRUE);
    Spell = 1; // Лед
}

if(m_App->m_Keyboard.GetKeyState(KEY_3) == TRUE) {
    m_App->m_Keyboard.SetLock(KEY_3, TRUE);
}

```



```

        Spell = 2; // Исцеление
    }

    if(m_App->m_Keyboard.GetKeyState(KEY_4) == TRUE) {
        m_App->m_Keyboard.SetLock(KEY_4, TRUE);
        Spell = 3; // Телепортация
    }

    if(m_App->m_Keyboard.GetKeyState(KEY_5) == TRUE) {
        m_App->m_Keyboard.SetLock(KEY_5, TRUE);
        Spell = 4; // Земляной вал
    }

    // Произносим заклинание, если скомандовали
    if(Spell != -1) {

```

Если заклинание произнесено, контроллер определяет, знает ли игрок это заклинание, достаточно ли у него маны для произнесения заклинания, и находится ли целевой персонаж в радиусе действия заклинания:

```

        // Произносим только известные заклинания
        // для которых достаточно маны

        if(g_PCChar->Def.MagicSpells[Spell/32] & (1 << (Spell & 31)) &&
            g_PCChar->ManaPoints >=
            m_App->m_SpellController.GetSpell(Spell)->Cost) {
            // Смотрим, на какого персонажа указали
            if((TargetChar = m_App->GetCharacterAt(
                m_App->m_Mouse.GetXPos(),
                m_App->m_Mouse.GetYPos())) != NULL) {
                // Не нацеливаемся на NPC
                if(TargetChar->Type != CHAR_NPC) {
                    // Получаем расстояние до цели
                    XDiff =
                        (float)fabs(TargetChar->XPos - Character->XPos);
                    YDiff =
                        (float)fabs(TargetChar->YPos - Character->YPos);
                    ZDiff =
                        (float)fabs(TargetChar->ZPos - Character->ZPos);
                    Dist = XDiff*XDiff + YDiff*YDiff + ZDiff*ZDiff;

                    // Смещаем расстояние на радиус цели
                    Range = GetXZRadius(TargetChar);
                    Dist -= (Range * Range);

                    // Получаем максимальную дистанцию заклинания
                    Range = GetXZRadius(Character);
                    Range +=
                        m_App->m_SpellController.GetSpell(Spell)->Distance;

                    // Выполняем заклинание только если цель
                    // в заданном диапазоне
                    if(Dist <= (Range * Range)) {

```

К данному моменту контроллер определил, какое заклинание должно быть произнесено. Вам надо сохранить координаты цели, номер произносимого заклинания и действие игрока в структуре, на которую указывает указатель **Character**:

```
// Установка данных заклинания
Character->SpellNum = Spell;
Character->SpellTarget = CHAR_MONSTER;

// Сохраняем координаты цели
Character->TargetX = TargetChar->XPos;
Character->TargetY = TargetChar->YPos;
Character->TargetZ = TargetChar->ZPos;

// Очищаем перемещение
(*XMove) = (*YMove) = (*ZMove) = 0.0f;

// Выполняем действия заклинания
SetAction(Character, CHAR_SPELL);

// Поворачиваемся к жертве
XDiff = TargetChar->XPos - Character->XPos;
ZDiff = TargetChar->ZPos - Character->ZPos;
Character->Direction =
    (float)atan2(XDiff, ZDiff);

// Устанавливаем действие
m_App->m_CharController.SetAction(Character,
                                   CHAR_SPELL);
}
}
}
```

Чтобы завершить обновление персонажа игрока, контроллер определяет, щелкнул ли игрок правой кнопкой мыши, что должно открывать окно статистики персонажа (путем помещения состояния статистики игрока в стек состояний):

```
// Помещаем кадр статистики, если нажата правая кнопка мыши
if(m_App->m_Mouse.GetButtonState(MOUSE_RBUTTON) == TRUE) {
    m_App->m_Mouse.SetLock(MOUSE_RBUTTON, TRUE);
    m_App->m_StateManager.Push(m_App->StatusFrame, m_App);
}

return TRUE;
}
```

Для использования производного класса контроллера персонажей, игра создает экземпляр класса **cChars** в объявлении **cApp**:

```
cChars m CharController;
```

Имея производный класс контроллера персонажей, вы готовы наследовать контроллер заклинаний. Держите глаза открытыми, мы пробежимся по нему очень быстро.

<b>ПРИМЕЧАНИЕ</b>	Производный класс контроллера заклинаний находится в файлах Game Spells.h и Game Spells.cpp.
-------------------	--

```

class cSpells : public cSpellController
{
    private:
        cApp *m_App;

    public:
        BOOL SetData(cApp *App) { m_App = App; return TRUE; }
        BOOL SpellSound(long Num);
};

```

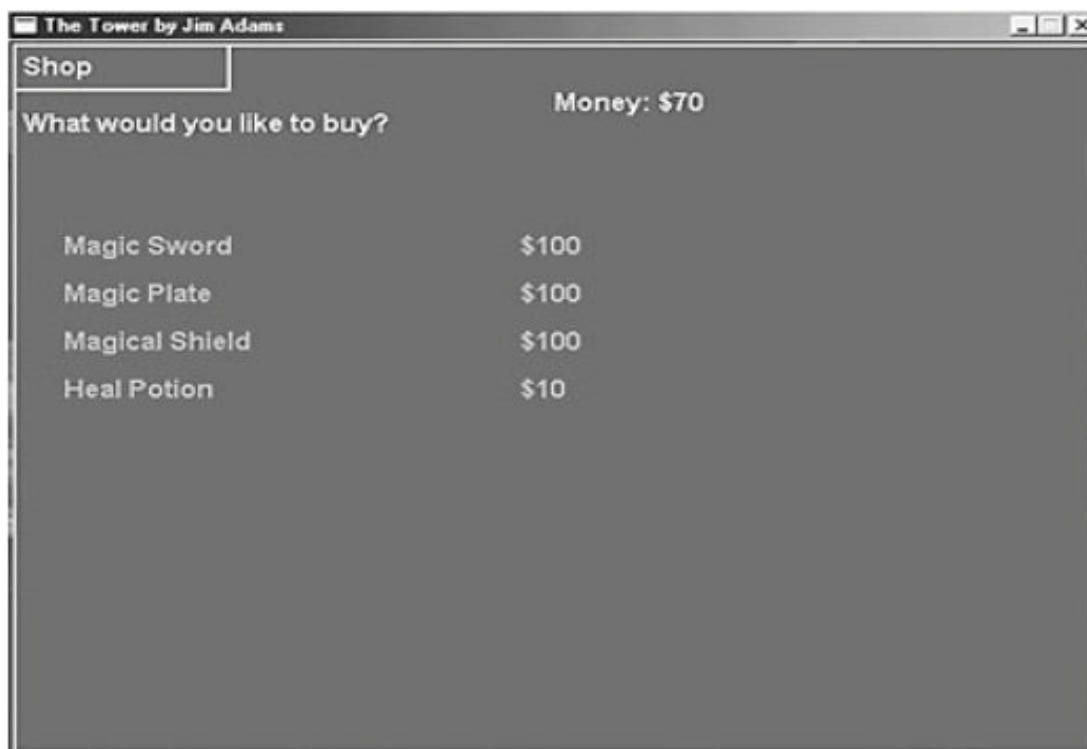
Подобно производному контроллеру персонажей, производный контроллер заклинаний имеет функцию **SetData**, сообщающую контроллеру, к какому классу приложения обращаться. В функции **SpellSound** вы используете указатель на приложение для вызова функции **cApp::PlaySound**.

Надеюсь вы не моргали, поскольку это все, что относится к производному классу контроллера заклинаний! Единственная переопределяемая функция — та, которая воспроизводит звук; остальная функциональность поддерживается базовым классом **cSpellController**. Чтобы использовать производный класс в игре, вы объявляете его экземпляр в объявлении класса приложения:

```
cSpells m_SpellController;
```

## Поддержка торговли

Ранее вы читали о том, как состояние **BarterFrame** используется для визуализации сцены торговли (рис. 16.8), где игрок может приобретать предметы у персонажа.



*Рис. 16.8. Интерфейс торговли слева отображает предметы для продажи, а в правом верхнем углу имеющееся количество денег для приобретения предметов*

Как это состояние узнает, какие предметы продаются? В игре имеется единственный способ включить состояние торговли — вызов его через срабатывание скрипта посредством действия скрипта «Торговля с персонажем». Это действие, в свою очередь, вызывает функцию **cApp::SetupBarter**, которая настраивает информацию, необходимую функции **BarterFrame**. Эта информация включает персонаж, который продает предметы, а также имя файла системы управления имуществом персонажа (ICS):

```

BOOL cApp::SetupBarter(sCharacter *Character, char *ICSFilename)
{
    g_BarterChar = Character;
    strcpy(g_BarterICS, ICSFilename);
    m_StateManager.Push(BarterFrame, this);

    return TRUE;
}

```

Функция состояния **BarterFrame** сканирует загруженную ICS, отображая каждый содержащийся в списке имущества персонажа предмет на экране. Если игрок щелкает по предмету, и у него достаточно денег, предмет покупается. Когда игрок завершает свои дела с лавочником, состояние торговли извлекается из стека состояний, и происходит возврат к игровому процессу.

## Воспроизведение звуков и музыки

Во время игры воспроизводится музыка и другие звуки. Эти игровые звуки, несколько плоховатые (можете сказать, что я не мастер звукозаписи!), воспроизводятся вызовом **PlaySound**. Единственный аргумент **PlaySound** — индекс в массиве имен звуковых файлов, который вы объявляете в начале кода приложения:

```

// Глобальные имена файлов звуковых эффектов
#define NUM_SOUNDS 9

char *g_SoundFileNames[NUM_SOUNDS] = {
    { "..\\Data\\Attack1.wav" },
    { "..\\Data\\Attack2.wav" },
    { "..\\Data\\Spell.wav" },
    { "..\\Data\\Roar.wav" },
    { "..\\Data\\Hurt1.wav" },
    { "..\\Data\\Hurt2.wav" },
    { "..\\Data\\Die1.wav" },
    { "..\\Data\\Die2.wav" },
    { "..\\Data\\Beep.wav" }
};

```

Заметьте, что количество имен звуковых файлов определяется макросом **NUM\_SOUNDS**. Вы должны гарантировать отсутствие попыток воспроизвести несуществующий звук, поскольку такая попытка приведет к краху системы. Для воспроизведения одного из допустимых звуков вы используете следующую функцию:

```
BOOL cApp::PlaySound(long Num)
{
    if(Num >=0 && Num < NUM_SOUNDS) {
        m_SoundData.Free();
        if(m_SoundData.LoadWAV(g_SoundFileNames[Num]) == TRUE)
            m_SoundChannel.Play(&m_SoundData);

        return TRUE;
    }

    return FALSE;
}
```

Функции **PlaySound** необходимо загрузить воспроизводимый звук, используя объект **cSoundData**. После этого звук воспроизводится из памяти. Почти также, как вы вызываете функцию **PlaySound**, можно воспроизводить различные песни, используя функцию **PlayMusic**. Функция **PlayMusic** также получает индекс в массиве имен музыкальных файлов, который определен следующим образом:

```
char *g_MusicFileNames[] = {
    { "..\\Data\\Cathedral_Sunrise.mid" },
    { "..\\Data\\Distant_tribe.mid" },
    { "..\\Data\\Escape.mid" },
    { "..\\Data\\Jungle1.mid" },
    { "..\\Data\\Magic_Harp.mid" },
    { "..\\Data\\Medi_Strings.mid" },
    { "..\\Data\\Medi_techno.mid" },
    { "..\\Data\\Song_of_the_sea.mid" },
    { "..\\Data\\Storm.mid" }
};
```

Здесь нам не надо отслеживать номер песни (мы живем на дикой стороне!), поэтому можно сразу перейти к функции **PlayMusic**:

```
BOOL cApp::PlayMusic(long Num)
{
    // Не меняем песню, если она уже воспроизводится
    if(g_CurrentMusic == Num)
        return TRUE;
```

Перед продолжением работы вы проверяете, какая песня воспроизводится в данный момент. Глобальная переменная отслеживает последнюю воспроизводившуюся песню, и, если эта песня продолжает воспроизводиться, вам не надо запускать воспроизведение той же самой песни снова (продолжается воспроизведение текущей песни). Если воспроизводится новая песня, громкость плавно понижается до нуля, текущая песня освобождается, загружается новая песня и запускается воспроизведение музыки:

```
    // Останавливаем и освобождаем текущую песню
    m_MusicChannel.Stop();
    m_MusicChannel.Free();

    // Плавно понижаем громкость музыки, давая DirectMusic
    // достаточно времени для завершения последней песни
    // (иначе новая песня не будет воспроизводиться корректно).
```

```

// Число 700 основывается на громкости воспроизведения музыки,
// можете подстроить его
DWORD Timer = timeGetTime() + 700;
while(timeGetTime() < Timer) {
    DWORD Level = (Timer - timeGetTime()) / 10;
    m_MusicChannel.SetVolume(Level);
}

// Загружаем и воспроизводим новую песню
m_MusicChannel.Load(g_MusicFileNames[Num]);
m_MusicChannel.Play(70, 0);

// Запоминаем номер новой песни
g_CurrentMusic = Num;

return TRUE;
}

```

## Визуализация сцены

В ходе игрового процесса в стек помещаются различные состояния — вы используете эти состояния чтобы определять, какую графику визуализировать в сцене. Общей частью большинства этих состояний является то, что их графика визуализируется поверх карты и персонажей. Для сохранения простоты вы создаете одну функцию, которая визуализирует на экране только задник и персонажи, оставляя рисование остальной графики функции состояния. Функция **RenderFrame** визуализирует карту и персонажей:

```

BOOL cApp::RenderFrame(long Elapsed)
{
    long i, j;

    // Визуализация упрощенной сетки для z-значений
    m_Graphics.EnableZBuffer(TRUE);
    m_SceneObject.Render();

    // Рисование фона (составленного из шести текстур)
    m_Graphics.EnableZBuffer(FALSE);
    m_Graphics.BeginSprite();
    for(i = 0; i < 2; i++) {
        for(j = 0; j < 3; j++)
            m_SceneTextures[i*3+j].Blit(j*256,i*256);
    }
    m_Graphics.EndSprite();

    // Рисование трехмерных объектов
    m_Graphics.EnableZBuffer(TRUE);
    m_CharController.Render(Elapsed);
    m_SpellController.Render();

    return TRUE;
}

```

Карта визуализируется идентично главе 9 (обратитесь к этой главе за дополнительной информацией об использовании движка, совмещающего двухмерную и трехмерную графику). Новое здесь — визуализация

персонажей. Вызов **cCharacterController::Render** обновляет анимацию персонажей и рисует соответствующие им сетки. Функция завершается вызовом для визуализации заклинаний.

## Обработка скриптов

Обработка скриптов управляет всем игровым содержимым. Это включает добавление персонажей на карту, отображение диалогов и другие функции, которые не запрограммированы жестко в движке игры.

Пример использует класс скрипта и производный класс скрипта, разработанные в главе 12. В то время, как класс скрипта хранится в файлах **script.h** и **script.cpp**, используемую производную версию класса скрипта игра **The Tower** хранит в файлах **game\_script.h** и **game\_script.cpp**. Пропустив класс скрипта (поскольку он остался таким же, как в главе 12), исследуем производный класс скрипта с именем **cGameScript**:

```
class cGameScript : public cScript
{
    private:
        // Внутренние массивы переменных и флагов
        BOOL m_Flags[256];
        long m_Vars[256];

        // Родительский объект приложения
        cApp *m_App;

        // Текстовое окно для отображения сообщений
        cWindow m_Window;
```

Скрипты используют массивы флагов и переменных (**m\_Flags** и **m\_Vars**), оба массива содержат по 256 элементов. Некоторые действия скриптов используют эти флаги и переменные для хранения данных и выполнения проверок условий с целью управления потоком обработки скрипта. Также сохраняется указатель на экземпляр класса приложения (для вызова функций приложения), и создается объект текстового окна для отображения диалогов персонажей и других текстов.

Затем в классе **cGameScript** вы определяете объект **sRoutePoint**, используемый скриптами для конструирования и назначения маршрутов персонажам:

```
// Маршрутная точка для конструирования маршрута персонажа
long m_NumRoutePoints;
sRoutePoint *m_Route;
```

Далее следует основная часть функций, являющихся функциями обработки действий скрипта. Эти функции вызываются когда начинается обработка действия из скрипта — например, функция **Script\_SetFlag** вызывается когда в скрипте обрабатывается действие установки флага. Взгляните на прототипы этих функций:

```
// Стандартные действия обработки
sScript *Script_End(sScript*);
sScript *Script_Else(sScript*);
sScript *Script_EndIf(sScript*);
sScript *Script_IfFlagThen(sScript*);
sScript *Script_IfVarThen(sScript*);
sScript *Script_SetFlag(sScript*);
sScript *Script_SetVar(sScript*);
sScript *Script_Label(sScript*);
sScript *Script_Goto(sScript*);
sScript *Script_Message(sScript*);

// Действия, связанные с персонажами
sScript *Script_Add(sScript*);
sScript *Script_Remove(sScript*);
sScript *Script_Move(sScript*);
sScript *Script_Direction(sScript*);
sScript *Script_Type(sScript*);
sScript *Script_AI(sScript*);
sScript *Script_Target(sScript*);
sScript *Script_NoTarget(sScript*);
sScript *Script_Bounds(sScript*);
sScript *Script_Distance(sScript*);
sScript *Script_Script(sScript*);
sScript *Script_CharMessage(sScript*);
sScript *Script_Enable(sScript*);
sScript *Script_CreateRoute(sScript*);
sScript *Script_AddPoint(sScript*);
sScript *Script_AssignRoute(sScript*);
sScript *Script_AlterHPMP(sScript*);
sScript *Script_Ailment(sScript*);
sScript *Script_AlterSpell(sScript*);
sScript *Script_Teleport(sScript*);
sScript *Script_ShortMessage(sScript*);
sScript *Script_Action(sScript*);
sScript *Script_IfExpLevel(sScript*);

// Действия торговли/обмена
sScript *Script_Barter(sScript*);

// Действия, связанные с предметами
sScript *Script_IfItem(sScript*);
sScript *Script_AddItem(sScript*);
sScript *Script_RemoveItem(sScript*);

// Действия, связанные с барьерами
sScript *Script_AddBarrier(sScript*);
sScript *Script_EnableBarrier(sScript*);
sScript *Script_RemoveBarrier(sScript*);

// Действия, связанные с триггерами
sScript *Script_AddTrigger(sScript*);
sScript *Script_EnableTrigger(sScript*);
sScript *Script_RemoveTrigger(sScript*);

// Действия, связанные со звуком
sScript *Script_Sound(sScript*);
sScript *Script_Music(sScript*);
sScript *Script_StopMusic(sScript*);

// Действие завершения игры
sScript *Script_WinGame(sScript*);
```



```
// Действия комментариев и разделителей
sScript *Script_CommentOrSeparator(sScript*);

// Действие ожидания
sScript *Script_Wait(sScript*);

// Генерация случайных чисел
sScript *Script_IfRandThen(sScript*);

// Принудительная визуализация кадра
sScript *Script_Render(sScript*);
```

Ух! Как много функций — и все они, как я сказал, непосредственно относятся к действиям скрипта. К счастью, функции обработки действий скрипта короткие и простые. В главе 10 были представлены скрипты, а в главе 12 представлен класс скрипта, так что обращайтесь в случае необходимости к этим главам, а сейчас вновь сосредоточимся на объявлении **cGameScript**:

```
// Функция обработки if/then
sScript *Script_IfThen(sScript *ScriptPtr, BOOL Skip);
```

Со всеми, относящимися к **if...then** функциями в шаблоне действий, проще разработать единую функцию, которая будет иметь дело с условной обработкой. Эта функция (**Script\_IfThen**) получает указатель на следующее действие скрипта после действия **if...then** и флаг, определяющий состояние условия. Если **Skip** установлен в **TRUE**, все последующие действия скрипта пропускаются, пока не будет найдено действие скрипта **Else** или **EndIf**, в то время как, если **Skip** установлен в **FALSE**, условие считается выполненным, и все действия скрипта обрабатываются, пока не будет найдено действие скрипта **Else** или **EndIf**. Заметьте, что действие скрипта **Else** меняет состояние флага **Skip** (с **TRUE** на **FALSE** и наоборот), обеспечивая правильную обработку конструкции **if...then...else**.

Объявление **cGameScript** завершается еще двумя закрытыми функциями — первая, **Release**, вызывается для освобождения внутренних данных скрипта, каждый раз когда завершается обработка скрипта. Вторая функция, **Process**, содержит большую инструкцию **switch**, отправляющую обрабатываемые действия скрипта соответствующим им функциям (точно также, как было показано в главе 12):

```
// Перегруженные функции обработки
BOOL Release();
sScript *Process(sScript *Script);

public:
    cGameScript();
    ~cGameScript();

    BOOL SetData(cApp *App);
    BOOL Reset();
```

```
    BOOL Save(char *Filename);  
    BOOL Load(char *Filename);  
};
```

Завершается **cGameScript** прототипами открытых функций — у вас есть конструктор, деструктор, функция **SetData**, сохраняющая указатель на экземпляр класса приложения, функция для сброса всех флагов и переменных, и две функции для сохранения и загрузки флагов и переменных в файл.

Вы вставляете объявление экземпляра производного класса скрипта **cGameScript** в объявление **cApp** для его использования главным приложением:

```
cGameScript m_Script;
```

Хотя объект **m\_Script** объявлен закрытым, большинство объектов в игре используют объект скрипта. Теперь причина объявления этих дружественных классов в объявлении класса приложения обрела смысл!

## Собираем части

Теперь вы хорошо знакомы с отдельными фрагментами головоломки. С примером игры на CD-ROM этой книги вы получите настоящий практический опыт совмещения всех этих частей вместе! Вы узнали о том, как компоненты определяются, разрабатываются и кодируются. С вызовом функции **cApp::Init**, за которым следуют повторяющиеся вызовы **cApp::Frame** игра оживает! Исполнение скриптов, взаимодействие персонажей, заклинания и атаки - все летает. Каждый компонент вносит свою лепту, и все они работают сообща, чтобы образовать целое.

Исследование проекта игры я рекомендую начать с файлов **WinMain.h** и **WinMain.cpp**; эти файлы содержат класс приложения, который формирует каркас приложения. Как подчеркивалось в этой главе, вы можете следовать потоку исполнения программы, от инициализации до завершения.

## Завершаем создание игры

Несмотря на размер и простоту, пример программы в этой главе демонстрирует потенциал использования информации, которой я поделился с вами в этой книге. Представьте себе, что вы сможете сделать с вашими игровыми проектами, применяя методы, показанные в этой и других главах. Понимая, как каждая часть работает индивидуально, и затем собирая эти части и понимая, как все они работают совместно, вы готовы создавать любые виды ролевых игр.

Используя библиотеки базовых функций для обработки графики, звука и ввода, ваш процесс создания игр может действительно ускориться. Больше не надо беспокоиться о низкоуровневых мелочах. Вместо этого приоритет отдается компонентам, используемым для управления игрой; вы можете

потратить свое время на управление персонажами, загрузку и рисование карт, и использование предметов и скриптов. Разрабатывая эти компоненты вы создаете свой собственный пример игры.

Ваш следующий шаг — взять пример игры и модифицировать его. Докопайтесь до корней. Найдите, как они работают. Измените скрипты, добавьте новых NPC и монстров, сделайте больше уровней. Когда вы будете точно понимать как и почему все это работает, вы поймете насколько легко проектировать и собирать жизнеспособную — и успешную — игру.

### **Программы на CD-ROM**

На прилагаемом к книге CD-ROM содержатся перечисленные ниже программы. Он содержит полный исходный код игры The Tower.

**The Tower** — Пример игры, The Tower, это заверченный игровой проект, взявший все, что вы узнали в этой книге, и собравший это воедино в простой для понимания пакет. Местоположение: \BookCode\Chap16\The Tower\.

# **Часть V**

## **Приложения**

### **Приложение А**

#### **Список литературы**

### **Приложение В**

#### **Содержимое CD-ROM**

### **Приложение С**

#### **Словарь терминов**



# Приложение А

## Список литературы

У каждого хорошего программиста есть хороший набор книг. Все программисты начинали с нуля, изучали основы, копили знания и продирались сквозь фрагменты кода. Наличие в вашем распоряжении хорошего набора книг — верный способ ускорить рост вашего программистского опыта.

В этом приложении я расскажу о моих любимых книгах по программированию, а также о некоторых замечательных Web-сайтах, которые я люблю посещать.

### Рекомендованное чтение

Я использую перечисленные ниже книги в своей программистской работе как общие справочники по рассмотренным в этой книге темам. Каждая книга, упомянутая в приложении, содержит информацию, относящуюся к материалам из этой книги.

#### Профессиональная анимация с DirectX

*Джим Адамс*

Premier Press, ISBN: 1-5920-0037-1

Эта книга (написанная мной!) содержит исключительно полезную информацию о техниках анимации, используемых в самых современных играх. Скелетная и морфирующая анимация, анимация лица и даже системы анимации тряпичных кукол — все это обсуждается в этой небольшой кладези знаний.

#### Dragon Magazine

Периодическое издание, Wizards of the Coast, Inc.

Это журнал, посвященный ролевым играм. В каждом выпуске содержатся исследования, истории, трюки и советы, письма и многое другое, относящееся к ролевым играм. Если вы фанат традиционных ролевых игр — это журнал для вас.

## **Dungeon Adventures**

Периодическое издание, Wizards of the Coast, Inc.

Это источник приключений для Dungeons & Dragons. Каждый выпуск содержит лучшие приключения, разработанные читателями и профессионалами по всему миру.

## **Справочник игрока Dungeons & Dragons, 3-е издание**

*Монти Кук, Джонатан Твит, Слип Вильямс*

Wizards of the Coast, Inc., 2000. ISBN: 0-7869-1550-1

Каждый серьезный игрок RPG слышал о большой франшизе Dungeons & Dragons — она во многом определила облик современного мира RPG. Эта книга представляет собой последнее издание правил игры.

## **Справочник мастера подземелий Dungeons & Dragons, 3-е издание**

*Монти Кук, Джонатан Твит, Слип Вильямс*

Wizards of the Coast, Inc., 2000. ISBN: 0-7869-1551-X

Эта книга, являющаяся компаньоном «Справочник игрока Dungeons & Dragons, 3-е издание», предназначена для мастеров подземелий (рефери игры). Если вы хотите узнать вселенную D&D, стоит приобрести эти книги.

## **Программирование изометрических игр с DirectX 7.0**

*Эрнест Пазера*

Premier Press, Inc., 2001. ISBN: 0-7615-3089-4

Книга необходима тем, кто серьезно интересуется техниками программирования изометрических игр. Хотя в моей книге и приводится общее описание использования изометрической графики, книга Эрнеста Пазеры раскрывает весь феномен изометрических игр в деталях.

## **lex & yacc**

*Джон Р. Левин, Тони Мэйсон, Дуг Браун*

O'Reilly & Associates, Inc., 1995. ISBN: 1-56592-000-7

В своей книге я описал использование простой системы скриптов, названной Mad Lib Scripting. Чтобы разработать более профессиональную систему скриптов, использующую похожий на программу код (что-то вроде C++),

вам потребуется копаться в таких темах, как лексический анализ и грамматические структуры. Я уверен, что в этом вам поможет книга «lex & уасс», описывающая основные этапы создания собственного скриптового языка и компилятора.

## **Программирование многопользовательских игр**

*Тодд Барон*

Premier Press, Inc., 2001. ISBN: 0-7615-3298-6

Как вы узнали из только что прочитанной книги, многопользовательские игры — важная тема. Книга Тодда Баррона — замечательный источник для получения информации о многопользовательских играх, которая осталась за рамками книги «Программирование ролевых игр с DirectX». Охватывая темы от основ функционирования сетей до создания настоящей многопользовательской игры, книга «Программирование многопользовательских игр» может оказаться именно той, которая вам нужна.

## **Программирование для Windows, 5-е издание**

*Чарльз Петзольд*

Microsoft Press, 1998. ISBN: 1-57231-995-X

Эта книга должна быть у каждого, кто серьезно занимается программированием для Windows. Охватывая практически все основы идеологии Windows, эта книга остается одним из наиболее часто используемых мной справочников.

## **Краткое руководство по написанию рассказов**

*Маргарет Льюк*

McGraw-Hill, 1999. ISBN: 0-07-039077-0

Верно. Даже при написании коротких рассказов следует соблюдать правильные пропорции, и эта книга является прямолинейным руководством по работе с основами каждой истории — разработка сюжета, структура рассказа и создание персонажей. Узнайте, что надо и что не надо делать при написании рассказов, как создать красочные персонажи — каждый с уникальными индивидуальностью и историей — как разработать уникальный сюжет и многое другое.

## **Мечи & схемы: руководство разработчика компьютерных ролевых игр**

*Нил Хэлфорд, Джэн Хэлфорд*

Premier Press, Inc., 2001. ISBN: 0-7615-3299-4



Взгляните на другую сторону создания игр - вопросы дизайна. Книга, которую вы читаете сейчас в основном фокусируется на вопросах программирования RPG (с небольшими вкраплениями тем о проектировании), поэтому, возможно, стоит приобрести экземпляр «Мечей & схем», где рассматриваются секреты мира проектирования RPG — от сожетных деревьев до игровых сценариев.

## **Дзен программирования игр с Direct3D**

*Питер Уэлш*

Premier Press, Inc., 2001. ISBN: 0-7615-3429-6

Кто не хотел бы книгу, посвященную исключительно Direct3D? Эта книга предоставляет детальный обзор возможностей, предоставляемых Direct3D. В то время, как моя книга дает только краткий тур по Direct3D (от использования графической системы до рисования трехмерных полигонов и сеток), «Дзен программирования игр с Direct3D» копает вглубь от основ Direct3D. Это подробное руководство для начинающих по программированию Direct3D.

## **Получение помощи в Web**

В наши дни многие люди (если не большинство) обращаются за информацией в Интернет. В этом приложении я привел список Web-сайтов, которые я использую для поиска информации по RPG, в том числе по программированию, разработке и игре.

## **Программирование ролевых игр с DirectX**

Я скромно начинаю с моего Web-сайта <http://home.att.net/~rpgbook>. На нем находится обновленная информация к этой книге, а также ресурсы для разработки и программирования ваших RPG. Вы можете следить за появляющимися на сайте новыми статьями, обновлениями кода и информацией о выпускаемых программах!

## **www.GameDev.net**

Это один из наиболее авторитетных сайтов для энтузиастов программирования. Направьте браузер в его сторону для получения статей, уроков, сообщений форумов, новостей, обзоров книг и многого другого. Я регулярно посещаю форум по DirectX, так что не стесняйтесь писать мне.

## **XTreme Games**

Домашняя страница Андре ЛаМота, автора нескольких популярных книг о программировании игр (и редактора серии Game Development издательства

Premier Press) является порталом XTremeGames, LLC. Цель этого сайта — помочь независимым разработчикам в публикации их игр, главным образом через продажу программного обеспечения. Если вы хотите присоединиться к игровой индустрии и у вас есть игра для публикации, посетите сайт <http://www.XGames3D.com>.

## **Flipcode**

Еще один замечательный ресурс для программистов. Он содержит новости, ресурсы, исходные коды и другие полезные вещи, которые держат людей приклеенными к экранам. Взгляните на все это по адресу <http://www.flipcode.com>.

## **Домашняя страница MilkShape 3-D**

Это домашняя страница MilkShape 3-D, бюджетной низкополигональной программы моделирования, которая находится на прилагаемом к книге CD-ROM. Посетите этот сайт для получения обновлений и плагинов, которые используются для импорта и экспорта моделей различных форматов (в том числе и X-файлов). Чтобы посетить домашнюю страницу MilkShape 3-D направьте ваш браузер на <http://www.swissquake.ch/chumbalum-soft/>.

## **Agetec**

Agetec Inc (сайт расположен по адресу <http://www.agetec.com>) является создателем замечательной программы RPG Maker для Sony PlayStation. С RPG Maker вы можете самостоятельно с нуля создать ролевую игру в стиле SNES. В программе есть все — графика, скрипты, боевые последовательности. Этот сайт хорошая отправная точка для тех, кто хочет создать RPG не углубляясь в дебри программирования.

## **Wizards of the Coast**

Это дом Wizards of the Coast, Inc., создателей Dungeons & Dragons. Посетите этот сайт (по адресу <http://www.Wizards.com>) для получения ресурсов дедушки всех RPG — D & D, и пока вы там, посмотрите всю линию их продуктов, относящихся к RPG.

## **White Wolf Publishing**

Это сайт создателей Vampire the Masquerade и ряда других систем RPG. Сайт (<http://www.white-wolf.com>) содержит великолепные ресурсы, чтобы начать путь во тьму (среди злобных созданий ночных дорог).

## **Steve Jackson Games**

Это домашняя страница системы GURPS Стива Джексона, расположенная по адресу <http://www.sjgames.com/gurps>. Компания Стива ответственна за

рождение таких известных игр, как GURPS, OGRE, Car Wars, Illuminati: New World Order и многих других. Посетите этот сайт, если вам нужна информация по Общецелевой Универсальной Ролевой Системе (Generic Universal Role-Playing System, GURPS), включая список книг и свободно загружаемые базовые правила.

## **Polycount**

Этот сайт заполнен моделями для различных трехмерных игр. Здесь вы можете найти художников и моделлеров для своего проекта. Чтобы посмотреть на модели перейдите по адресу <http://www.polycount.com>. Убедитесь, что у вас установлен MilkShape 3-D, который понадобится для редактирования моделей.

## **RPG Planet**

Заполненный обзорами, статьями и обсуждениями последних новостей компьютерных RPG, этот сайт (расположенный по адресу <http://www.rpgPlanet.com>) является обязательным местом паломничества для всех фанатов RPG! Если вы на компьютерной стороне ролевых игр и хотите следить за последними событиями индустрии RPG, регулярно посещайте этот сайт.

## **RPG Host**

Являясь площадкой для размещения многих, посвященных RPG Web-сайтов, RPG Host предоставляет большой каталог, относящихся к RPG материалов. На сайте вы найдете игры, ресурсы для загрузки, новости, форумы — и все это относится к вашему любимому жанру игр. Загляните на RPG Host по адресу <http://www.rpgghost.com> и, пока вы там, посмотрите огромный список размещенных сайтов, относящихся к RPG.

## **[www.gamedev.net/reference/articles/frpg/site](http://www.gamedev.net/reference/articles/frpg/site)**

Хотя я уже упоминал GameDev.net, этот раздел содержит множество информации о дизайне RPG. Сюда определенно надо заглянуть тем, кто нуждается в небольшой помощи или руководстве. Поддерживает этот сайт Крис Беннет из Dwarfsoft.

# Приложение В

## Содержимое CD-ROM

Прилагаемый к книге компакт-диск содержит примерно 650 000 000 упорядоченных по спирали байт данных, называемых обычно CD. Для просмотра содержимого диска ничего устанавливать не надо; а значит на ваш жесткий диск будут установлены или скопированы только те файлы, которые вы выберете сами. Просматривать содержимое компакт-диска можно в любой операционной системе, поддерживающей отображение HTML-файлов с графикой, однако не на все из этих систем можно будет установить находящиеся на диске программы.

Если у вас включена функция автозапуска, то после помещения компакт-диска в привод ваш браузер автоматически отобразит HTML-интерфейс. Если автозапуск отключен, вы можете получить доступ к содержимому компакт-диска, выполнив следующие действия:

1. Вставьте компакт-диск в привод CD-ROM вашего компьютера и закройте лоток.
2. Откройте окно **My Computer** (Мой компьютер) или **Windows Explorer** (Проводник) и щелкните по значку привода CD-ROM.
3. Найдите и откройте файл `start_here.html` (он подходит для большинства браузеров). Будет показано лицензионное соглашение Premier Press.
4. Прочитайте лицензионное соглашение. Если вы согласны с ним, щелкните по кнопке **I Agree**, чтобы принять лицензию и перейти к пользовательскому интерфейсу. Если вы не согласны с лицензионным соглашением, щелкните по кнопке **I Disagree**. Работа с CD будет прекращена. Окно пользовательского интерфейса Premier Press содержит кнопки навигации и область содержимого. Кнопки навигации расположены в левой части окна браузера. Вы можете перемещаться по пользовательскому интерфейсу Premier Press, щелкая соответствующие кнопки. В результате загружаются страницы, чье содержимое отображается в правой части экрана. В следующих разделах описывается содержимое компакт-диска.

## DirectX 9.0 SDK

Поскольку эта книга называется «Программирование ролевых игр с DirectX» и DirectX постоянно упоминается во всей книге, на CD-ROM находится полная версия DirectX 9.0 SDK.

Один из лучших способов ближе познакомиться с DirectX — изучение примеров программ и исходных кодов, которые Microsoft предоставляет вместе с SDK. В частности, взгляните на демонстрационные программы Direct3D; они научат вас многому о программировании трехмерных игр.

Также в SDK включена справочная документация DirectX. Если вы еще не использовали справочную систему DirectX API, вы быстро поймете, что это один из главных ваших друзей, когда дело касается разработки трехмерной графики. Не используйте один только алфавитный указатель. Уделите внимание и книгам на вкладке **Contents**, поскольку они содержат ценную информацию об архитектуре и возможностях в целом (а также специфическую информацию о каждом компоненте).

## Пробная версия GoldWave 5

Программа GoldWave (выпущенная GoldWave Inc.) — это полнофункциональный редактор звуковых файлов, предоставляющий, помимо прочего, следующие возможности:

- Многодокументный интерфейс для редактирования нескольких файлов в одном сеансе.
- Возможность редактировать большие файлы (размером до 1 Гбайт).
- Настройка редактирования в ОЗУ (для скорости) или на жестком диске (для больших файлов).
- Получение графиков в реальном времени (амплитуда, спектр, полоса и спектрограммы).
- Отдельное, с изменяемым размером, окно Device Controls для доступа к аудиоустройствам.
- Быстрая перемотка вперед и назад.
- Набор спецэффектов (искажение, доплер, эхо, фильтры, механизация, смещение, панорамирование, выравнивание громкости, инвертирование, повторная выборка, эквалайзер, подавление шумов, изменение масштаба времени, задание основного тона и т.д.).
- Поддержка многих форматов файлов (WAV, MP3, OGG, AIFF, AU, VOX, MAT, SND, VOC, необработанные двоичные данные и текстовые данные) с возможностью преобразовывать один формат в другой.
- Интерфейс с поддержкой перетаскивания данных.

- Непосредственное редактирование волновых форм с помощью мыши.

## Пробная версия Paint Shop Pro 8

На компакт-диске находится 30-дневная пробная версия Paint Shop Pro (от Jasc Software, Inc.), одного из лучших инструментов для создания, редактирования и ретуширования изображений. Paint Shop Pro — это мощная программа рисования, предоставляющая вам все необходимые инструменты для создания текстур и фоновых изображений.

## gameSpace Light

Программа gameSpace Light (от Caligari Corporation) широко используется художниками по моделям и аниматорами и получил признание индустрии за передовые возможности, такие как визуализация с учетом гибридной излучательности и прямые манипуляции с пользовательским интерфейсом. gameSpace Light предоставляет возможности, удовлетворяющие потребности двух новых рынков: рынка дизайна и создания трехмерного содержимого для Web.



# Приложение С

## Словарь терминов

**ASCII.** Сокращение для American Standard Code for Information Interchange. Это код, в котором числа от 0 до 255 означают буквы, цифры, знаки препинания и другие символы. Код ASCII стандартизирован, чтобы облегчить передачу текста между компьютерами или между компьютером и периферийными устройствами.

**D&D.** Сокращенное название игры Dungeons & Dragons.

**DirectInput, DI.** Компонент DirectX, отвечающий за работу с устройствами ввода (такими, как клавиатура, мышь или джойстик).

**DirectMusic, DM.** Компонент DirectX, используемый для воспроизведения музыки и звуковых файлов (таких, как MIDI и волновые файлы).

**DirectPlay, DP.** Компонент DirectX, используемый для сетевой функциональности.

**DirectSound, DS.** Компонент DirectX, используемый для воспроизведения цифровых звуков.

**DirectX.** Являющийся детищем Microsoft игровой API, позволяющий программисту создавать игры, не беспокоясь о деталях, связанных с особенностями использования установленного оборудования.

**DirectX Audio.** Начиная с DirectX 8 (и также в DirectX 9), DirectX Audio представляет комбинацию всех звуковых компонентов, а именно DirectSound и DirectMusic.

**DirectX Graphics.** Начиная с DirectX 8, вся графическая функциональность из DirectDraw и Direct3D была объединена в единый компонент, названный DirectX Graphics. Это верно и для DirectX 9.

**GURPS.** Аббревиатура для Общей универсальной системы ролевых игр (Generic Universal Role-Playing System).

**IP-адрес (IP-address).** Сетевой адрес, представляемый в виде четырех чисел, например, 192.168.1.2.

**Mad Lib Script, MLS.** Термин, который я выдумал чтобы описать метод создания скриптов, где вы применяете заранее написанные действия,



использующие возможность выбора формата для получения требуемых данных.

**MIDI.** Формат хранения музыки.

**Unicode.** 16-битовые символы, обеспечивающие возможность кодирования всех известных символов.

**Z-буфер (Z-buffer).** Массив значений, определяющих глубину (в сцене) каждого пикселя.

**Автоматическая карта (Auto map, auto-mapping).** Функция игрового движка для автоматического отслеживания и отображения ранее посещенных областей игровых карт и уровней. Эта возможность позволяет игроку видеть, где он уже был.

**Алгоритм художника (Painter's algorithm).** Порядок, в котором визуализируются объекты. Объекты визуализируются на основании расстояния до них от зрителя, от дальних к ближним.

**Альфа-канал (Alpha channel).** Непрозрачность изображения определяется альфа-значениями для каждого пикселя, чередующимся с цветовыми компонентами (например, ARGB), альфа-значениями для каждого пикселя, хранящимися в отдельной альфа-поверхности, или постоянным альфа-значением для всей поверхности.

**Альфа-проверка (Alpha testing).** Режим визуализации, который пропускает рисование полностью прозрачных пикселей.

**Альфа-смешивание (Alpha blending).** Комбинирование цветов или альфа-значений в ходе визуализации.

**Антагонист (Antagonist).** Персонаж (или персонажи), вносящий в сюжет хаос и беспорядок, обычно противостоящий попыткам героя (протагониста) достичь поставленных целей и замыслов. *См. также* Протагонист.

**Атрибуты (Attributes).** Определяют рост и способности персонажа в игре. Эти атрибуты (и способности) могут варьироваться от физических особенностей, таких как вес, до измеряемых оценок силы.

**Аудио-путь (Audio path).** Управляет потоком данных от объекта исполнителя, сегментов, синтезатора и звуковых буферов.

**Базовый класс (Base class).** Используется для описания родителя производного класса.

**Барьер (Barrier).** Препятствие на карте, останавливающее перемещение персонажа.

**Библиотека (Library).** В программировании библиотека это набор программных функций, сгруппированных в единую сущность. Вы используете библиотеки посредством API.

**Блок** (Tile). Блок графических пикселей, используемый для совместного формирования больших изображений.

**Боевые аранжировки** (Battle arrangements). Заранее определенные местоположения персонажей для боевых сцен.

**Большие растры** (Big bitmap). Заранее визуализированные большие растровые изображения уровней, используемые в специально приспособленных графических движках.

**Броня** (Armor). Обобщенное описание любой части экипировки, которую игрок может надеть для увеличения его уровня защиты против атак. Это может быть что угодно — от частей доспехов до щитов и ботинок.

**Буфер атрибутов** (Attribute buffer). Массив значений, используемых объектом **ID3DXMesh** для визуализации сеток. Каждый элемент массива соответствует отдельному полигону сетки. Каждый элемент хранит идентификационный номер материала, используемого при визуализации соответствующей грани.

**Веер треугольников** (Triangle fan). Список полигонов, созданных из набора соединенных вершин вокруг центральной вершины.

**Венгерская нотация** (Hungarian notation). Соглашение об оформлении кода в котором префикс переменной или функции определяет тип данных.

**Вершина** (Vertex). Отдельная точка в n-мерном пространстве. В трехмерной графике вершина содержит три координаты (X, Y и Z), определяющие ее пространственное местоположение.

**Вершинный шейдер** (Vertex shader). Набор директив, определяющих как обрабатываются и рисуются вершины.

**Видимость класса** (Class visibility). Объявление данных и функций класса, ограничивающее или запрещающее доступ к указанным функциям и данным.

**Визуализация** (Render). Рисование объекта или объектов.

**Виртуальный код клавиши** (Virtual key code). Код, используемый Microsoft Windows, и близко напоминающий скан-код.

**Врезки** (Cut-scenes). Временные перерывы в ходе сюжета; используются для смены сцены.

**Вторичная сеть** (Pass-along network). Сеть или маршрутизатор, которые передают сетевые данные другой сети или маршрутизатору.

**Вторичный буфер** (Backbuffer). Невидимая поверхность, на которой можно рисовать растры и другие изображения, пока первичный буфер отображает текущее изображение.

**Вторичный звуковой буфер** (Secondary sound buffer). Буфер, содержащий звуковую волну.

**Выборка (Sample).** Одиночное измерение амплитуды звуковой волны.

**Выкачиватель сообщений (Message pump).** Функция, которая постоянно вытягивает сообщения из очереди сообщений приложения и обрабатывает их.

**Главный список заклинаний (Master spell list, MSL).** Список используемых в игре заклинаний.

**Главный список персонажей (Master character list, MCL).** Список используемых в игре персонажей.

**Главный список предметов (Master item list, MIL).** Список используемых в игре предметов.

**Глобальная очередь (Global queue).** Очередь сообщений, ожидающих обработки операционной системой.

**Глобальный уникальный идентификатор (Global Unique Identification, GUID).** Гарантированно уникальный номер, размером 128 бит.

**Действие (Action).** Команда скрипта.

**Дерево (Tree).** Это не большое, красивое покрытое листвой растение, а структура, содержащая узлы, соединенные с другими узлами в древоподобной манере. Подумайте о реальном дереве — ствол дерева разделяется на сучья, сучья делятся на ветки, а на ветках растут листья. Все эти точки (ствол, сучья, ветви, листья) называются узлами.

**Деструктор (Destructor).** Функция, вызываемая при уничтожении объекта.

**Децибел (Decibels, dB).** Единица сравнения между двумя уровнями интенсивности звука.

**Дополнительные статусы (Status ailments).** Физические или ментальные условия которые увеличивают или уменьшают способности и атрибуты персонажа.

**Дочерний шаблон/фрейм (Child template/frame).** Шаблон (или фрейм), относящийся к другому шаблону.

**Друг класса (Friend class).** Разрешает объявленному другом классу свободный доступ к данным и функциям класса.

**Жанр (Genre).** Стиль или классификация. Например, ролевые игры и стратегии это два различных жанра игр.

**Загружаемые звуки (Downloadable sounds, DLS).** Оцифрованные звуки (или наборы звуков), которые загружаются в синтезатор.

**Задержка (Latency).** Время, которое тратится сетевым сообщением на путешествие от источника до места назначения. Чем больше задержка, тем больше времени требуется сообщению чтобы достигнуть места назначения, и это обычно увеличивает время запаздывания в процессе игры.

**Запаздывание (Lag).** Задержка между временем, когда действие было запрошено, и временем, когда оно произошло. Запаздывание обычно связывается с сетевыми играми, в которых сеть настолько загружена, что команды игроков обрабатываются гораздо дольше, чем обычно. Результатом является медлительное управление игрой.

**Затухание (Falloff).** Уменьшение интенсивности света с увеличением расстояния.

**Игровая система (Gaming system).** Набор правил, управляющих ролевой игрой.

**Игровой движок (Game engine).** Набор программного кода и функций, запускающих фактическое игровое приложение.

**Излом (Plot point).** Точка важного изменения сюжета.

**Иерархия фреймов (Frame hierarchy).** Категоризированный список фреймов.

**Иерархия шаблонов (Template hierarchy).** Структурированный список шаблонов.

**Индексированный буфер вершин (Indexed vertex buffer).** Буфер вершин, хранящий вершины в произвольном порядке, использующий массив индексов, чтобы определить, какие вершины образуют полигоны.

**Инди (Indies).** Независимые разработчики.

**Интерфейс программирования приложений (Application Programming Interface, API).** Посредник в мире программирования. API предоставляет программисту интерфейс к лежащей ниже функциональности отдельного программного кода.

**Искусственный интеллект (Artificial Intelligence, AI).** Описывает интеллектуальность или управление обработкой независимых от игрока персонажей игры с целью имитации определенного поведения или действий.

**Исполнитель (Performance).** Главный объект, используемый для управления воспроизведением музыкального объекта.

**Исходное событие (Inciting incident).** Событие, приведшее к основным событиям, управляющим сюжетом.

**Камера (Camera).** В графике это представление свободно перемещаемой точки просмотра, используемой для отображения карт и уровней.

**Каркас приложения (Application framework).** Структура приложения (в отношении планировки и потока обработки).

**Карта (Map).** Место или места, где разворачивается игра.

**Карта текстур (Texture map).** Растровое изображение, наносимое на поверхность полигона, для улучшения визуального представления рисуемых полигонов.

**Класс (Class).** В программировании набор программного кода и данных, уникальных для экземпляра объекта. В ролевых играх класс это категория или профессия персонажа (такая как разведчик, воин или маг).

**Класс персонажа (Character class).** Классификация персонажей, определяемая проектом игры.

**Клиент (Client).** Клиентское приложение — это программа, через которую игрок взаимодействует с игрой в ходе сетевой игровой сессии.

**Ключ (Key).** Ориентация объекта в конкретный момент времени. Используется в анимационных последовательностях.

**Ключевой кадр (Key frame).** Последовательность ключей, используемая в анимации.

**Коллекции инструментов (Instrument collections).** Наборы инструментов, используемых для воспроизведения музыки.

**Кольцевой буфер (Circular buffer).** Буфер замкнутый в кольцо, где конец буфера соединен с его началом.

**Компилятор (Compiler).** Программа, используемая для преобразования вашего программного кода в исполняемое приложение.

**Компиляция (Compile).** Процесс преобразования вашего программного кода в другую форму (такую, как двоичный машинный язык), пригодную для исполнения или дальнейшего использования.

**Консоль (Console).** Термин имеет два значения. Во-первых, специальный экран позволяющий пользователям модифицировать, читать или набирать игровую информацию или сообщения. Во-вторых, это домашняя игровая система, такая как Sony PlayStation.

**Конструктор (Constructor).** Метод класса, вызываемый непосредственно при создании объекта класса, чьей целью является инициализация членов класса в заданное состояние.

**Копирование с учетом прозрачности (Transparent blit).** Операция рисования, пропускающая прозрачные пиксели.

**Корневой узел (Root node).** Первый узел в списке, к которому присоединены все другие узлы.

**Критическая секция (Critical section).** Объект, используемый для синхронизации потоков в рамках одного процесса.

**Кульминация (Climax).** Точка наивысшего напряжения сюжета.

**Лобби-сервер (Lobby server).** Объект сетевого сервера, управляющий множественными подключениями, и направляющий пользователей к игровым сессиям.

**Локальное пространство** (Local space). Локальная система координат трехмерного объекта.

**Магия** (Magic). Описывает паранормальные способности, которые могут использоваться играющими персонажами.

**Маршрутизатор** (Data router). Устройство, направляющее сетевые данные.

**Массив индексов** (Index array). Массив значений, по которым конструируется серия полигональных граней.

**Мастер игры** (Game Master, GM). См. Мастер подземелья.

**Мастер подземелья** (Dungeon master, DM). В игре Dungeons & Dragons это судья, управляющий всеми игровыми аспектами и контролирующий ход игры.

**Материал** (Material). Описывает вид и текстуру трехмерного объекта; используется для задания параметров визуализации объекта, включая цвет, рельеф и отраженные блики.

**Матрица вида** (View matrix). Матрица, представляющая преобразование вида.

**Матрица мирового преобразования** (World transformation matrix). Матрица, представляющая мировое преобразование.

**Матрица проекции** (Projection matrix). Матрица используемая для конвертирования из непреобразованных координат в преобразованные координаты.

**Матричное произведение** (Matrix concatenation). Комбинация двух матриц.

**Механизм регулирования** (Throttling mechanisms). Механизм, ограничивающий поток входящих и исходящих сетевых данных.

**Мировое преобразование** (World transformation). Преобразование из локальных координат объекта в мировые координаты.

**Мировое пространство** (World space). Трехмерные координаты, ориентированные относительно начала координат игрового мира.

**Многопоточность** (Multithreading). Обработка нескольких потоков.

**Модель** (Model). Сетка и назначенные материалы, используемые при визуализации сетки.

**Модель компонентных объектов** (Component Object Model, COM). Форма модульного программирования, принятая Microsoft. Является сердцем компонентов DirectX.

**Модификатор** (Modifier). Значение, которое меняет другое значение.

**Модификаторы** (Patches). Инструменты.

**Модульность** (Modular). Модульные программы или библиотеки могут быть быстро и легко вставлены в проекты. Обычно модульные библиотеки

состоят из повторно используемых функций, таких как процедуры работы с устройствами ввода.

**Монстр** (Monster). Название дается любому игровому персонажу, выступающему против игрока. Не путайте монстров с независимыми персонажами. У монстров почти нет других дел в игре, кроме сражений, в то время как независимые персонажи важны для сюжета игры и ее прохождения.

**Набор GM/GS** (GM/GS set). Набор инструментов General MIDI/General Synth.

**Набор боевых правил** (Combat rule set). Правила, управляющие сражениями.

**Навыки** (Skills). Похожи на атрибуты в том, что описывают способности персонажа. К навыкам может относиться все, что угодно, от способности взбираться на стены до умения вести переговоры в напряженных ситуациях.

**Направленный свет** (Directional light). Свет, распространяющийся в заданном направлении.

**Наследование** (Inheritance). В программировании описывает способность производного класса поглощать функциональность его базовых классов (включая функции и переменные).

**Настраиваемый формат вершин** (Flexible vertex format, FVF). Описание содержимого данных вершины, от цвета вершины до ее координат.

**Небесный куб** (Sky box). Текстурированный трехмерный куб, окружающий зрителя.

**Независимый персонаж** (Non-player character, NPC). Любой персонаж игры, которым не может управлять игрок. NPC может быть кем угодно - от дружелюбного лавочника до самого ничтожного демона из подземелий.

**Непреобразованные координаты** (Untransformed coordinates). Трехмерные координаты.

**Номер модификатора** (Patch number). Идентификационный номер инструмента.

**Нормаль** (Normal). Вектор, описывающий направление, в котором обращена вершина, плоскость, свет или полигон.

**Обработка на основе состояний** (State-based processing, SBP). Структура или поток исполнения функций, базирующийся на текущем установленном состоянии.

**Ограничивающий параллелепипед** (Bounding box). Прямоугольная область, полностью заключающая в себя объект.

**Ограничивающая сфера** (Bounding sphere). Сферическая область, полностью заключающая в себя объект.

- Одноранговая сеть** (Peer-to-peer). Прямое сетевое соединение одного компьютера с другим.
- Окно** (Window). Прямоугольная графическая область, относящаяся к приложению. Применяется для отображения выходных данных приложения.
- Оружие** (Weapon). Предметы, которые могут использоваться для атаки (такие, как мечи, ножи, дубины и камни).
- Основной темп** (Master tempo). Темп, используемый звуковой системой в ходе воспроизведения музыки.
- Отбрасывание невидимых граней** (Backface culling). Удаление (или пропуск) полигонов (которые направлены от точки просмотра) в процессе визуализации.
- Отражения** (Specular). Цветные блики на освещенном объекте.
- Отслеживание траектории** (Dead reckoning). Вид обновления значения на основании экстраполяции из набора известных значений.
- Очередь сообщений приложения** (Application message queue). Очередь, которая хранит сообщения Windows, относящиеся к приложению.
- Очки маны** (Mana points, MP). Подобно очкам повреждений, очки маны описывают уровень магической мощи персонажа или количество магии, которое персонаж может использовать. Обычно очки маны уменьшаются с каждым произнесенным заклинанием, но позже восстанавливаются после отдыха или специальных действий.
- Очки опыта** (Experience points, EXP). Описывает (в численном выражении) рост продвижения опытности персонажа (подобно уровню опыта).
- Очки повреждений** (Hit points). Описывают либо уровень здоровья персонажа, либо количество повреждений, которое может получить персонаж прежде чем умрет.
- Первичный звуковой буфер** (Primary sound buffer). Главный объект, содержащий звук перед началом его воспроизведения звуковой системой.
- Перегрузка функций** (Function overloading). Метод предоставления нескольких прототипов одной и той же функции, отличающихся друг от друга набором аргументов.
- Перелистывание** (Flip). Переключение видимой и невидимой частей изображения.
- Перемещение** (Translating). Действия по передвижению объектов.
- Персонаж игрока** (Player character, PC). Персонаж, находящийся под управлением игрока (обычно это главный герой игры).



**Пирамида видимого пространства** (Frustum, viewing frustum). Видимая часть окружающей зрителя сцены, содержащаяся внутри шести плоскостей (расположенных в форме усеченной пирамиды).

**Плоскости отсечения** (Clipping planes). Удаление объектов из конвейера визуализации на основании того, с какой стороны (передней или задней) относительно каждой из плоскостей находится этот объект.

**Подмножество** (Subset). Набор полигонов сетки, сгруппированных по соответствующему материалу.

**Поле зрения** (Field of view, FOV). Видимая часть окружающей зрителя сцены.

**Полигон** (Polygon). Замкнутая фигура, образованная группированием вершин, соединенных ребрами. Полигоны являются основой трехмерной графики (поскольку вся трехмерная графика визуализируется как полигоны). Полигон может быть величиной с один пиксель, а может занимать весь экран.

**Полиморфизм** (Polymorphism). Способность производного класса обращаться к функциям его базового класса так, будто производный класс тот же самый, что и базовый.

**Полоса пропускания** (Bandwidth). Мера количества данных, которые могут быть переданы через сетевое соединение.

**Полоса треугольников** (Triangle strip). Список полигонов, созданных из серии последовательно определенных вершин.

**Помощники** (Supporting character). Персонажи, поддерживающие по сюжету протагониста.

**Порт** (Port). Виртуальный причал, к которому направляются входящие сетевые данные. Порты последовательно нумеруются, обычно в диапазоне от 0 до 60 000. Данные, предназначенные конкретному порту IP-адреса, направляются прямо в этот порт, так что никакие другие порты не могут мешать этим данным.

**Поставщик услуг** (Service provider). Сетевой протокол.

**Построение персонажа** (Character building). Процесс увеличения способностей и атрибутов персонажа по ходу развития игры.

**Поток** (Thread). Отдельный процесс выполнения.

**Поток вершин** (Vertex stream). Трубопровод, по которому данные вершин передаются от буфера данных к визуализатору.

**Поток программы** (Program flow). Поток, согласно которому программа выполняется или структурирована.

**Потоковое аудио** (Streaming audio). Большие звуковые данные неэффективно хранить в памяти. Воспроизведение больших звуков вовлекает последовательное воспроизведение небольших фрагментов, что создает

впечатление непрерывности воспроизведения. Процесс чтения звуковых данных по фрагментам и называется потоковым аудио.

**Предзнаменование** (Foreshadowing). Взгляд назад, чтобы ввести причину возникновения эпизода сценария.

**Преобразование** (Transformations). Вычисления, используемые для модификации координат.

**Преобразование вида** (View transformation). Преобразование, ориентирующее вершины вокруг точки просмотра.

**Преобразование проекции** (Projection transformation). Преобразование конвертирующее непреобразованные координаты в преобразованные координаты.

**Преобразование фрейма** (Frame transformation). Преобразование, применяемое к фрейму.

**Преобразованные координаты** (Transformed coordinates). Координаты, подходящие для дисплея.

**Проект** (Project). В программировании проектом называется собранный воедино набор кода, библиотек и других, относящихся к программе, вещей.

**Проектный документ** (Design document). Библия игрового проекта; содержит развернутое описание создаваемой игры.

**Прожектор** (Spotlight). Источник, испускающий свет в виде конуса и освещающий только предметы, расположенные внутри конуса.

**Производный класс** (Derived class). Класс, заимствующий структуру базового класса, и расширяющий его функциональность путем добавления или модификации существующих функций и данных.

**Пространство модели** (Model space). См. Локальное пространство.

**Протагонист** (Protagonist). Герой истории или персонаж вокруг которого вращается сюжет.

**Протокол TCP/IP** (Transfer Control Protocol/Internet Protocol). Сетевой протокол передачи данных, используемый для передачи данных через Интернет.

**Протокол UDP** (User Datagram Protocol). Форма сетевой передачи, в которой не отслеживается успешное получение сетевых данных клиентом.

**Процедура оконных сообщений** (Window message procedure). Функция приложения, обрабатывающая оконные сообщения.

**Процесс** (Process). Однопоточный блок исполнения. Вы можете думать о вашей игре как о процессе, который может быть разбит на другие процессы, такие как процесс ввода, процесс визуализации графики и т.д.

**Рабочее пространство** (Workspace). Будучи похожим на стол, но виртуальный, рабочее пространство управляет всеми относящимися к проекту

материалами (такими, как файлы исходного кода, ресурсы и библиотеки), группируя их в списки для простой навигации и изменения в соответствии с потребностями проекта.

**Развязка** (Denouement). Заключительная часть сценария.

**Раса** (Race). Как и в реальной жизни, персонажи ролевых игр относятся к различным расам (например, люди, эльфы или даже огры).

**Растровое изображение** (Bitmap). Шаблон пикселей, формирующих большое изображение. Также английский термин используется как название принадлежащего Microsoft формата хранения изображений (обозначаемого расширением файла .BMP).

**Расширенный ASCII** (Extended ASCII). Расширение стандарта ASCII, определяющее использование 256 символов, вместо установленного в ASCII 128-символьного лимита.

**Рекурсивная функция** (Recursive function). Функция, которая во время исполнения вызывает сама себя.

**Ретроспектива** (Flashback). Взгляд назад во времени в виде врезки в сценарии.

**Ролевая игра** (Role-playing game, RPG). Игра, в которой игроки принимают роли воображаемых персонажей.

**Сегмент** (Segment). Музыкальный фрагмент.

**Сервер** (Server). Сетевое приложение, служащее в качестве центрального обрабатывающего узла игрового мира. Клиентские приложения подключаются к серверам и начинают пересылку действий игроков туда и обратно. Однако, именно сервер выполняет обработку большей части игровой функциональности.

**Сессия** (Session). Время работы с игрой.

**Сетка** (Mesh). Сгруппированные полигоны.

**Сеть** (Network). Несколько компьютеров, соединенных друг с другом для обмена данными и совместной работы.

**Система живой ролевой игры** (Live Action Role-Playing System, LARPS). Используется для описания любой игровой системы, в которой игроки наряжаются и разыгрывают свои роли, также как в театральной постановке. Одна из таких игр Vampire, the Masquerade.

**Система управления имуществом** (Inventory control system, ICS). Движок, используемый для управления имуществом персонажей или списком объектов карты.

**Скалярное произведение** (dot product, scalar product). Произведение длины двух векторов и угла между ними.

KdZg-h^6FDFORGHAgZq\_gb\_ ihkueZ\_fh\_ deZ\bZImjhc b  
 deZ\brZ [ueZ gZ`ZIZ beeb hlms\_gZ  
 Kd\_e\_lgZy k6NdzQPHVGK K\_ldZ f\_gyxsZy k\hx nhjfm khh  
 e\_`Zs\_fm \ \_\_ hkgh\ gZ[hjm \hh[jZ`Z\_fuo dhkl\_c  
 Kdjbil 6FULSW GZ[hj ijh]jZffguo bgkljmdpbc h[jZ  
 b]jh\h]h ^\b`dZ <u bkihevam\_l\_ kdjbilu ^ey baf\_g  
 g\_h[oh^bfhklb i\_j\_ijh]jZffbjh\Zgby b]jh\h]h ^\b`dZ  
 Keh`ghklbRPSOLF DWLRQV Ij\_iylkl\by klhysb\_ gZ im  
 Kh]eZr\_gby h[ hnhjfe\_gkRGLRZQWHRQQGZ[hj ijZ\be be  
 hij\_^\_e\_gbc j\_]eZf\_glbjmxsbo nhjfZI beeb kljmdlmj  
 Kha^Zg&UHDWXXUHKu\Z\_l g\_^jm`\_ex[guc beeb g\_jZamf  
 Wlh \dexqZ\_l ex[uo i\_jkhgZ`\_c k dhlhjufb \u fh`\_l  
 dZd kd\_e\_lu beeb`bZd bgo dZd ehrZ^b  
 Khjlbjh\dZ ih ]emHSGVRUWLQhKhjlbjh\dZ h[t\_dlh\ gZ hkgh  
 ]em[bgu \ kp\_g\_  
 KhklhygbWDDWVWh khklhygby baf\_g\_gby beeb khklhyg  
 dhgdj\_lguc fhf\_gl ijh]jZffZ fh`\_l gZoh^blvky \  
 khklhygbb dhlhj\_h fh`\_l [ulv i\_j\_dexq\_gh gZ ^jm]h  
 nhjfm h[jZ[hldb dhlhjhc ke\_^m\_l b]jZ Fh`\_l\_ ^mf  
 h kb]gZeZo k\\_lhnjhZ khklhygb\_ djZkgh]h k\\_lZ h  
 fh]ml ^\b]Zlvky khklhygb\_ a\_e\_gh]h k\\_lZ hagZq  
 \_oZlv  
 Kibkhd bfms\_klQZYHQVORUVWKibkhd h[t\_dlh\ ijbZg^e\_  
 i\_jkhgZ`m beeb dZjl\_  
 Kibkhd Ij\_m]hevg7ULHDQOOWHWKibkhd iheb]hgh\ kha  
 ]jmiibjh\dhc \\_jrbg ih Ijb  
 Kihkh[ghk\$E L O L W K l f H ] b [ m l u  
 KijZcl6StuL K\h[h^gh i\_j\_f\_sZ\_fuc [ehd ]jZnbq\_kdbo i  
 ij\_^klZ\eyxsbc b]jh\u\_ i\_jkhgZ`b b h[t\_dlu  
 KjZ`\_gb&RPEDW HibkZgb\_ ijhba\hevghc ihke\_^h\Z  
 ijb\h^ysbo d ZIZd\_ beeb aZsbl\_ i\_jkhgZ`Z b]jhdZ  
 Kkuehqugu\_ rZ[7HhPuSODHWOUF L QNhjfZ mdZaZgby gZ beeb  
 rZ[ehgu  
 KlZ^by l\_dklmjbjhVZh[WyXUWHDJMVIZi \ dhg\\_c\_j\_ \bamZ  
 hij\_^\_eyxsbc dZd dZjlZ l\_dklmj bkihevam\_lky ijb jh  
 KlZg^ZjlgZy k6WDDZQPHVGKfK\_ldZ  
 L\_dklmjgZy ]jm7HJZWXXUHRXS Iheb]hgu k]jmiibjh\Zggu  
 khhl\\_lkl\mxs\_c bf l\_dklmj\_

**Технический движок** (Tech engine). Движок, управляющий техническими аспектами игры, такими как визуализация графики, воспроизведение звука и обработка ввода.

**Точечный свет** (Point light). Объект, излучающий свет во всех направлениях.

**Точка просмотра** (Viewpoint). Точка, в которой находится зритель.

**Триггер** (Trigger). Размещенный на карте объект, запускающий выполнение скрипта, когда игрок прикасается к нему.

**Убийца игроков** (Player killer, PK). Персонаж, занимающийся грязной работой, убивая персонажи других игроков.

**Узел** (Node). Отдельная сущность, содержащаяся в дереве. *См. также* Дерево.

**Уровень** (Level). Либо уровень опыта персонажа, достигнутый в ходе игрового процесса, либо игровая карта, которую населяют и исследуют персонажи.

**Уровень кооперации** (Cooperative level). Параметр, определяющий как будет осуществляться совместный доступ к устройству или объекту.

**Уровень опыта** (Experience level). Используется для отслеживания главных уровней продвижения персонажа. Например, у персонажа может быть 10000 очков опыта, которые будут рассматриваться как уровень 5. Обычно каждое продвижение в уровне опыта приносит различные призы, такие как изучение новых заклинаний или умений.

**Фильтрация текстур** (Texture filtering). Техника, используемая для изменения пикселей текстурированных полигонов перед началом визуализации.

**Финальная бета** (Final beta). Последняя тестовая версия программного продукта перед его выпуском.

**Фоновый свет** (Ambient light). Постоянный уровень освещенности, равномерно освещающий все объекты сцены.

**Фрейм** (Frame). Используется в файлах .X для группировки связанных шаблонов с целью упрощения доступа.

**Цветовой ключ** (Color key). Цвет, представляющий прозрачные пиксели. Пиксели, окрашенные в тот же цвет, что и цветовой ключ, пропускаются в ходе визуализации.

**Цель вызова** (Calling purpose). Причина вызова функции состояния.

**Частица** (Particle). Представляет собой произвольный свободно перемещающийся объект, используемый в графике для улучшения визуального качества игры. Частицы могут применяться для представления дымовых следов, искр света и даже неразумных жучков бесцельно порхающих вокруг.

**Частота выборки** (Sampling rate). Частота с которой записывается цифровой звук.

**Шаблон** (Template). Предопределенная структура конкретных данных. Например, заявление о приеме на работу можно рассматривать как шаблон, поскольку оно содержит заранее написанный текст с пустыми полями, которые заполняет соискатель.

**Шаблон действий** (Action template). Список действий скрипта и их соответствующих структур и использования.

**Шаблон фрейма** (Frame template). Шаблон, который содержит фрейм.

**Широкие символы** (Wide characters). См. Unicode.

**Щит** (Billboard). Полигон (или группа полигонов) автоматически выравниваемых, чтобы быть перпендикулярными линии взгляда игрока.

**Экранное пространство** (Screen space). Двухмерная система координат аналогичная экранным координатам.

**Ядро игры** (Gaming core). Библиотека функций, управляющих базовыми аспектами игрового программирования от обработки графики и рисования до обработки пользовательского ввода от таких устройств, как мышь и клавиатура.



# **Алфавитный указатель**