

СТАНИСЛАВ ГОРНАКОВ



DirectX 9

УРОКИ ПРОГРАММИРОВАНИЯ

НА C++

ПРОГРАММИРОВАНИЕ
ТРЕХМЕРНОЙ ГРАФИКИ

КОМПОНЕНТЫ Direct3D,
DirectInput, DirectMusic
И DirectSound

МАТРИЧНЫЕ
ПРЕОБРАЗОВАНИЯ

ПРОГРАММИРОВАНИЕ
ВЕРШИННЫХ И
ПИКСЕЛЬНЫХ ШЕЙДЕРОВ

СПРАВОЧНИК
ПО DirectX 9 SDK



PRO
ПРОФЕССИОНАЛЬНОЕ
ПРОГРАММИРОВАНИЕ



Станислав Горнаков

DirectX 9

УРОКИ ПРОГРАММИРОВАНИЯ НА C++

Санкт-Петербург

«БХВ-Петербург»

2004

УДК 681.3.068+800.92C++
ББК 32.973.26-018.1
Г67

Горнаков С. Г.

Г67 DirectX 9: Уроки программирования на C++. — СПб.:
БХВ-Петербург, 2004. — 400 с.: ил.
ISBN 5-94157-482-7

Рассмотрено профессиональное программирование трехмерной графики под Windows на языке C++ с использованием библиотеки DirectX 9. Раскрыты возможности компонента Direct3D по выводу трехмерной графики, текстурированию объектов, работе с освещением, вершинными и пиксельными шейдерами и др. Описаны также компоненты DirectInput, DirectMusic и DirectSound. Материал изложен в виде уроков и поможет читателю самостоятельно изучить технологию DirectX, на основе которой создаются профессиональные компьютерные игры. Прилагаемый компакт-диск содержит примеры, рассмотренные в книге.

Для программистов

УДК 681.3.068+800.92C++
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	<i>Екатерина Кондукова</i>
Зам. главного редактора	<i>Игорь Шишигин</i>
Зав. редакцией	<i>Григорий Добин</i>
Редактор	<i>Елена Яковлева</i>
Компьютерная верстка	<i>Натальи Смирновой</i>
Корректор	<i>Наталья Першакова</i>
Дизайн серии	<i>Инны Тачиной</i>
Оформление обложки	<i>Игоря Цырульников</i>
Зав. производством	<i>Николай Тверских</i>

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 22.10.04.

Формат 70×100¹/₁₆. Печать офсетная. Усл. печ. л. 32,25.

Тираж 3000 экз. Заказ №

"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02
от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-482-7

© Горнаков С. Г., 2004
© Оформление, издательство "БХВ-Петербург", 2004

Содержание

Введение.....	1
О чем эта книга.....	1
Что вы должны знать.....	3
Какое программное обеспечение мы будем использовать	3
Благодарности.....	3
Урок 1. Основы программирования под Windows.....	5
Создаем проект в Visual C++ .NET	5
Общая модель сообщений в Windows.....	7
Главная функция <i>WinMain</i>	8
Разбираем класс <i>windowsclass</i>	9
Создаем окно.....	13
Обрабатываем события в Windows	15
Главный обработчик событий.....	16
Итоги урока	20
Урок 2. DirectX 9.....	21
Компоненты DirectX 9.....	22
HAL.....	23
COM	23
Интерфейс <i>IUnknown</i>	24
Direct3D 9.....	26
Построение сцены в Direct3D	27
Интерфейсы Direct3D 9.....	28
Создание указателя на интерфейс.....	29
Установка DirectX 9 SDK.....	30
Итоги урока	33
Урок 3. Инициализация Direct3D.....	35
Создаем функцию для инициализации Direct3D	36
Обработка ошибок в DirectX	47
Рендеринг, или рисуем в окне приложения	48
Освобождаем ресурсы, захваченные Direct3D	51
Итоги урока	56
Урок 4. Рисуем 2D-объект	57
Установка формата вершин	58
Создание буфера вершин	60

Рендеринг объекта.....	66
Рисуем квадрат	74
Итоги урока	76
Урок 5. Матрицы.....	77
Сложение и вычитание матриц	77
Умножение матриц	78
Единичная матрица.....	79
Матрицы в Direct3D	79
Мировая матрица	80
Матрица вида.....	82
Матрица проекции	83
Итоги урока	86
Урок 6. Вывод на экран 3D-объекта	87
Матрицы преобразования	90
Рисуем куб	100
Индексация вершин.....	103
Итоги урока	114
Урок 7. Буфер глубины или Z-буфер.....	115
Связываем стороны куба.....	120
Итоги урока	130
Урок 8. Свет и материал	131
Свет.....	133
Нормаль.....	135
Установка света и материала	139
Итоги урока	150
Урок 9. Текст в Direct3D	151
Создание шрифта	151
Вывод текста на экран.....	156
Полноэкранный режим	158
Итоги урока	169
Урок 10. Текстурирование	171
Загрузка текстуры.....	172
Рендеринг текстурированного объекта	174
Итоги урока	188
Урок 11. Мультитекстурирование	189
Цветовые ключи	191
Итоги урока	194

Урок 12. Загрузка X-файлов	195
Итоги урока	210
Урок 13. Вершинные и пиксельные шейдеры	211
Графический конвейер	211
Фиксированный конвейер.....	213
Программируемый конвейер	214
Шейдеры	214
Вершинные шейдеры	216
Архитектура вершинных шейдеров	216
Синтаксис команд	217
Пиксельные шейдеры	227
Архитектура пиксельного шейдера.....	227
Синтаксис команд	229
Пишем пиксельный шейдер.....	230
Итоги урока	232
Урок 14. Инициализация DirectInput.....	233
Интерфейсы	234
Функции DirectInput8	235
Создание основного интерфейса.....	235
Создание устройства ввода.....	237
Установка формата данных устройства ввода.....	237
Установка уровня взаимодействия устройства ввода	238
Захват устройства ввода.....	238
Получение данных от устройства ввода	238
Буферизированные данные	239
Итоги урока	239
Урок 15. Работа с клавиатурой.....	241
Создание основного объекта DirectInput8.....	241
Создание устройства клавиатуры	242
Установка формата данных клавиатуры	243
Установка уровня взаимодействия клавиатуры	244
Захват доступа к клавиатуре.....	245
Получение данных с клавиатуры.....	245
Освобождение захваченных ресурсов	250
Итоги урока	251
Урок 16. Мышь	253
Создание устройства	254
Установка формата данных.....	255

Установка уровня взаимодействия	255
Захват доступа к мыши	256
Получение данных	256
Освобождение захваченных ресурсов	257
Построение класса <i>MyInput</i>	258
Описание функций класса <i>MyInput</i>	259
Итоги урока	262
Урок 17. DirectMusic	263
Интерфейсы DirectMusic	263
Создание приложения	264
Инициализация COM	265
Создание главного интерфейса	265
Создание загрузчика	266
Инициализация аудиосистемы	267
Загрузка файла из каталога	268
Загрузка сегмента в синтезатор и его воспроизведение	270
Освобождение захваченных ресурсов	271
Итоги урока	273
Урок 18. DirectSound	275
Интерфейсы DirectSound	276
Создание основного объекта	277
Установка уровня взаимодействия	277
Запись в созданный буфер	281
Воспроизведение данных	283
Классы	284
Класс <i>CSound</i>	284
Класс <i>CSoundManager</i>	286
Класс <i>CStreamingSound</i>	287
Класс <i>CWaveFile</i>	288
Итоги урока	289
Урок 19. Итоги	291
Создание приложения при помощи мастера	293
Сгенерированные классы	296
Файлы D3DApp.h и D3DApp.cpp	296
Файлы Urok19.h и Urok19.cpp	297
Файл D3DFont.h и D3DFont.cpp	300
Файлы D3DFile.h и D3DFile.cpp	301
Файлы DMUtil.h и DMUtil.cpp	304
Итоги урока	306
Заключение	307

ПРИЛОЖЕНИЯ.....309**Приложение 1. Справочная информация311**

DirectX Graphics	312
Интерфейсы DirectX Graphics	312
Функции DirectX Graphics	313
<i>Direct3DCreate9()</i>	313
<i>IDirect3D9::CreateDevice</i>	313
<i>IDirect3D9::GetAdapterDisplayMode</i>	314
<i>IDirect3DDevice9::Clear</i>	314
<i>IDirect3DDevice9::CreateVertexBuffer</i>	315
<i>IDirect3DDevice9::CreateVertexDeclaration</i>	315
<i>IDirect3DDevice9::CreateVertexShader</i>	316
<i>IDirect3DDevice9::DrawPrimitive</i>	316
<i>IDirect3DDevice9::GetDirect3D</i>	316
<i>IDirect3DDevice9::LightEnable</i>	317
<i>IDirect3DDevice9::Present</i>	317
<i>IDirect3DDevice9::SetLight</i>	318
<i>IDirect3DDevice9::SetMaterial</i>	318
<i>IDirect3DDevice9::SetRenderState</i>	318
<i>IDirect3DDevice9::SetStreamSource</i>	318
<i>IDirect3DDevice9::SetTexture</i>	319
<i>IDirect3DDevice9::SetTextureStageState</i>	319
<i>IDirect3DDevice9::SetTransform</i>	320
<i>IDirect3DDevice9::SetVertexDeclaration</i>	320
<i>IDirect3DDevice9::SetVertexShaderConstantF</i>	320
<i>IDirect3DBaseTexture9::SetPrivateData</i>	321
<i>IDirect3DVertexBuffer::Lock</i>	321
<i>ID3DXBaseEffect::SetVertexShader</i>	322
<i>ID3DXBaseMesh::DrawSubset</i>	322
<i>ID3DXFont::DrawText</i>	322
<i>ID3DXSkinInfo::SetFVF</i>	323
<i>D3DXAssembleShader()</i>	323
<i>D3DXAssembleShaderFromFile()</i>	323
<i>D3DXAssembleShaderFromResource()</i>	324
<i>D3DXCleanMesh()</i>	325
<i>D3DXColorModulate()</i>	325
<i>D3DXCompileShader()</i>	326
<i>D3DXCompileShaderFromFile()</i>	326
<i>D3DXCompileShaderFromResource()</i>	327
<i>D3DXComputeNormalMap()</i>	328
<i>D3DXCreateCubeTexture()</i>	329
<i>D3DXCreateCubeTextureFromFile()</i>	329
<i>D3DXCreateFont()</i>	330
<i>D3DXCreateMeshFVF()</i>	330
<i>D3DXCreateSPMesh()</i>	331

<i>D3DXCreateTextureFromFile()</i>	331
<i>D3DXCreateTextureFromFileEx()</i>	331
<i>D3DXDeclaratorFromFVF()</i>	333
<i>D3DXGetImageInfoFromFile()</i>	333
<i>D3DXLoadMeshFromX()</i>	333
<i>D3DXMatrixIdentity()</i>	334
<i>D3DXMatrixInverse()</i>	334
<i>D3DXMatrixLookAtLH()</i>	335
<i>D3DXMatrixMultiply()</i>	335
<i>D3DXMatrixMultiplyTranspose()</i>	336
<i>D3DXMatrixPerspectiveFovLH()</i>	336
<i>D3DXMatrixRotationX()</i>	337
<i>D3DXMatrixRotationY()</i>	337
<i>D3DXMatrixRotationZ()</i>	337
<i>D3DXMatrixScaling()</i>	337
<i>D3DXMatrixTranslation()</i>	338
<i>D3DXMatrixTranspose()</i>	338
<i>D3DXSaveTextureToFile()</i>	339
<i>D3DXVec3Normalize()</i>	339
Структуры DirectX Graphics	339
<i>D3DADAPTER_IDENTIFIER9</i>	339
<i>D3DBOX</i>	340
<i>D3DCOLORVALUE</i>	341
<i>D3DDISPLAYMODE</i>	341
<i>D3DLIGHT9</i>	342
<i>D3DMATERIAL9</i>	343
<i>D3DPRESENT_PARAMETERS</i>	344
<i>D3DRANGE</i>	345
<i>D3DRECT</i>	346
<i>D3DVECTOR</i>	346
<i>D3DVERTEXELEMENT9</i>	346
<i>D3DVIEWPORT9</i>	347
<i>D3DXMATERIAL</i>	347
<i>RECT</i>	348
Типы DirectX Graphics	348
<i>D3DBLEND</i>	348
<i>D3DBLENDOP</i>	349
<i>D3DCULL</i>	350
<i>D3DDEVTYPE</i>	350
<i>D3DLIGHTTYPE</i>	351
<i>D3DPOLL</i>	351
<i>D3DPRIMITIVETYPE</i>	352
<i>D3DRENDERSTATETYPE</i>	353
<i>D3DVERTEXBLEND_FLAGS</i>	361
<i>D3DZBUFFERTYPE</i>	361

Макросы DirectX Graphics	362
<i>D3DCOLOR_ARGB</i>	362
<i>D3DCOLOR_COLORVALUE</i>	362
<i>D3DCOLOR_RGBA</i>	363
<i>D3DTS_WORLDMATRIX</i>	363
DirectInput.....	363
Интерфейсы DirectInput.....	363
Функции DirectInput.....	364
<i>DirectInput8Create()</i>	364
<i>IDirectInput8::CreateDevice</i>	364
<i>IDirectInput8::EnumDevices</i>	365
<i>IDirectInput8::FindDevice</i>	365
<i>IDirectInput8::GetDeviceStatus</i>	366
<i>IDirectInput8::Initialize</i>	366
<i>IDirectInput8::RunControlPanel</i>	366
<i>IDirectInputDevice8::Acquire</i>	366
<i>IDirectInputDevice8::BuildActionMap</i>	367
<i>IDirectInputDevice8::GetCapabilities</i>	367
<i>IDirectInputDevice8::GetDeviceInfo</i>	367
<i>IDirectInputDevice8::GetDeviceState</i>	367
<i>IDirectInputDevice8::SetDataFormat</i>	368
<i>IDirectInputDevice8::SetCooperativeLevel</i>	368
Структуры DirectInput.....	369
<i>DIDATAFORMAT</i>	369
<i>DIMOUSESTATE</i>	369
DirectMusic.....	370
Интерфейсы DirectMusic.....	370
Функции DirectMusic.....	370
<i>IDirectMusicLoader8::LoadObjectFromFile</i>	370
<i>IDirectMusicPerformance8::InitAudio</i>	371
<i>IDirectMusicPerformance8::PlaySegmentEx</i>	371
<i>IDirectMusicSegment8::SetRepeats</i>	372
DirectSound	372
Интерфейсы DirectSound.....	373
Функции DirectSound	373
<i>DirectSoundCreate8()</i>	373
<i>IDirectSound8::CreateSoundBuffer</i>	373
<i>IDirectSound8::SetCooperativeLevel</i>	374
<i>IDirectSoundBuffer8::Lock</i>	374
<i>IDirectSoundBuffer8::Unlock</i>	375
<i>IDirectSoundBuffer8::Play</i>	375
Структуры DirectSound	376
<i>DSBUFFERDESC</i>	376
<i>WAVEFORMATEX</i>	376

Приложение 2. Web-ресурсы	378
Приложение 3. Список использованных источников.....	379
Приложение 4. Описание компакт-диска	380
Предметный указатель	381

Введение

Технология DirectX от корпорации Microsoft сегодня является стандартом программирования игр под платформу Windows. Более 90% компьютерных игр пишутся на языке C++ с использованием библиотеки DirectX. Со времен появления первой графической, и действительно мощной операционной системы Windows 95, корпорация Microsoft пытается выпускать мультимедийные библиотеки, посредством которых можно было бы заниматься программированием компьютерных игр. Самый первый пакет назывался Game SDK. Он был выпущен примерно в тот период, когда Windows 95 завоевывала весь мировой рынок. Game SDK был полностью основан на растровой графике, но это и вполне понятно, в тот момент просто не было мощных 3D-адаптеров, также имелась поддержка звука и устройств ввода. Год спустя выходят одна за другой версии DirectX 2 и DirectX 3, в их составе появляется Direct3D. После выхода Windows 98 библиотека DirectX пополнилась пятой и шестой версией. И только в 1999 году появилась DirectX 7, ознаменовавшая новую эпоху трехмерных игр. Через год выходит DirectX 8, после чего все сомнения по поводу того, на чем и с чем писать компьютерные игры, разваливаются как карточный домик. В середине 2003 года выпускается новая версия DirectX 9 и на данный момент это самая последняя версия, о которой и пойдет речь в этой книге.

О чем эта книга

Вся книга построена на уроках по DirectX 9 с использованием языка программирования C++. Уроки необходимо изучать в хронологическом порядке; без понимания предыдущего урока последующий урок будет практически недоступен для освоения.

На *первом уроке* раскрываются основы программирования под Windows, где в частности будет создано оконное приложение, в которое в дальнейшем будет интегрироваться исходный код каждого урока.

Второй урок даст вам ответ, что же такое DirectX, из каких компонентов состоит рассматриваемая библиотека, на основе какой из технологий строится DirectX. Соответственно будут получены необходимые сведения по COM и большой вводный курс в Direct3D, также будет изучена общая концепция построения сцены в Direct3D, а в конце всего урока подробно рассмотрена инсталляция DirectX 9 SDK на ваш компьютер.

На *третьем уроке* будет рассмотрена самая ключевая информация начального этапа работы с Direct3D — это инициализация Direct3D, создание устройства Direct3D и настройка аппаратной части компьютера, необходимой для работы. К слову сказать, все уроки до тринадцатого включительно будут посвящены Direct3D.

На *четвертом уроке* мы построим и выведем на экран простой треугольник с помощью буфера вершин и конвейера рендеринга, получая при этом основную информацию о геометрии сцены в Direct3D.

Матрицам посвящен *пятый урок*. Если вы забыли либо не знали, что такое матрицы, как сложить и перемножить их между собой, этот урок для вас. Матрицы в Direct3D имеют свое специфическое значение, об этом и многом другом весь пятый урок.

Шестой урок будет ознаменован выводом на экран полноценного трехмерного объекта с помощью концепции индексации вершин.

Буфер глубины — это тема *седьмого урока*. Сортировка точек объекта в зависимости от расстояния до глаз — это то, для чего необходим буфер глубины, имеющий еще название Z-буфер.

Восьмой урок посвящен материалу и свету. Затрагивается тема модели освещения, дается представление об источниках света, раскрывается общий принцип взаимодействия материала с освещением объекта, на основе которого строится общее восприятие цвета глазом человека. Нормали, необходимые для расчета освещения, тоже обсуждаются на этом уроке.

Текст, выводимый на экран, имеет огромное значение в играх, об этом весь *девятый урок*. В конце урока будет осуществлен переход из оконного режима в полноэкранный.

Десятый урок раскрывает основы текстурирования. Рассматривается стандартный способ наложения текстур на объект.

Одиннадцатый урок является продолжением темы о текстурировании. Будет изучено мультитестурирование.

На *двенадцатом уроке* в приложение будет загружена модель, сделанная сторонним 3D-редактором и конвертированная в X-формат для загрузки.

Тринадцатый урок посвящен вводу курсора в вершинные шейдеры. Этот большой теоретический урок раскрывает общую концепцию работы вершинных шейдеров, затрагивая вершинные шейдеры первой версии.

Три последующие урока — *четырнадцатый*, *пятнадцатый* и *шестнадцатый* — рассказывают об устройствах ввода, представленных интерфейсом DirectInput. Рассматривается работа с клавиатурой и мышью.

Уроки семнадцатый и *восемнадцатый* посвящены звуку. Для этих целей в DirectX существуют интерфейсы DirectMusic и DirectSound.

И самый последний урок — *девятнадцатый* — это итоговый урок по работе с Direct3D, DirectInput, DirectMusic. Будет создан ряд отдельных классов, более реально отражающих технику создания и компоновки компьютерных игр.

В конце всей книги вас ждет приятный сюрприз, материализованный в виде *приложения 1* — справочника по основным компонентам DirectX 9 SDK, где описаны наиболее часто используемые интерфейсы, структуры, функции, макросы.

В *приложении 2* приведены интересные интернет-ссылки. В содержании компакт-диска поможет разобраться *приложение 3*. А список использованных источников информации можно увидеть в *приложении 4*.

Что вы должны знать

Книга рассчитана на начинающего программиста, но предполагается, что вы уже знаете в минимальном объеме язык программирования C++ и можете создать проект в среде Visual C++ .NET, написать несколько строк кода, откомпилировать его и запустить полученный exe-файл. Все уроки и примеры, приведенные в данной книге, описываются в очень доступной форме. Однако, эта книга не о языке программирования C++ или о среде Visual C++ .NET.

Какое программное обеспечение мы будем использовать

При написании этой книги использовались: Windows XP, Visual C++ .NET, DirectX 9 SDK. Весь написанный код должен компилироваться и в более ранних версиях Visual C++, но будем исходить из того, что раз все вышеперечисленные программные продукты уже доступны в последних версиях, то соответственно они и использовались. Что касается других языков и библиотек программирования, то не вступая в полемику с людьми, пользующимися чем-то другим, хочу сказать только одно — 90% игр пишутся на C++ и DirectX. Если хотите зарабатывать на своих знаниях — учите то, на что сейчас делается ставка. Удачи вам!

Благодарности

Эту книгу я хочу посвятить любви всей моей жизни, которая постоянно меня подбадривала и оказывала большую моральную помощь в создании книги. Без моей жены этой книги не было бы вообще. Света, это все для тебя!

Конечно, хочу вспомнить своих родителей, без которых трудно себе представить появление этой книги и меня, как человека.

И еще хотелось бы поблагодарить издательство "БХВ-Петербург", обеспечившее самые благоприятные условия для написания этой книги, и лично отметить Анатолия Адаменко, Игоря Шишигина и Вадима Александровича Сергеева. Особая благодарность редактору книги Яковлевой Елене Сергеевне за большую проделанную работу.

Урок 1



Основы программирования под Windows

Прежде чем мы начнем программирование графики с помощью DirectX 9, нам необходимо создать каркас программы, являющийся своего рода рабочей поверхностью для трехмерного приложения. Ни для кого не секрет, что Windows многозадачная система, и в целом все то, что вы видите на экране монитора — это обыкновенное оконное приложение. Создав каркас программы, мы можем перейти к программированию графики, рисуя любые объекты в созданном приложении. Поэтому первоочередной нашей задачей будет написание кода окна, на основе которого и будут строиться все дальнейшие примеры к урокам. Что касается размеров оконного приложения, то они тоже играют роль. Имеются два режима — полноэкранный и оконный. В оконном режиме вы можете задавать размеры создаваемого окна, определять его стили и место вывода на экране. В полноэкранном режиме указываются разрешение и частота обновления экрана. На первых уроках мы будем использовать оконный режим, а в дальнейшем перейдем к полноэкранному, переход между двумя режимами сам по себе несложен.

Создаем проект в Visual C++ .NET

Весь этот урок будет посвящен созданию оконного приложения. Для этого в Windows, а точнее в среде программирования, которую мы используем, существуют стандартные средства. Начнем с того, что откроем среду программирования Visual C++ .NET и создадим проект. Я уверен, что вы знаете, как это делается, но на всякий случай повторим. Для создания проекта необходимо осуществить следующие действия.

1. Выполните команду **File | New | Project**. Появится диалоговое окно **New Project**.
2. В поле **Project Types** выберите папку **Visual C++ Projects**, а в поле **Templates** — **Win 32 Project**.

3. В поле **Name** укажите имя всего проекта. Чтобы не путаться, предлагаю назвать его **Urok1**. В поле **Locations** задайте директорию, в которой будут храниться ваши исходные коды, например **C:\Code**. Нажмите кнопку **OK**.

Появится новое диалоговое окно **Win 32 Application Wizard - Urok1** (рис. 1.1), в котором нужно перейти на вкладку **Application Settings** и установить в области **Additional options** флажок **Empty project**. В области **Application type** необходимо выбрать переключатель **Windows application**.

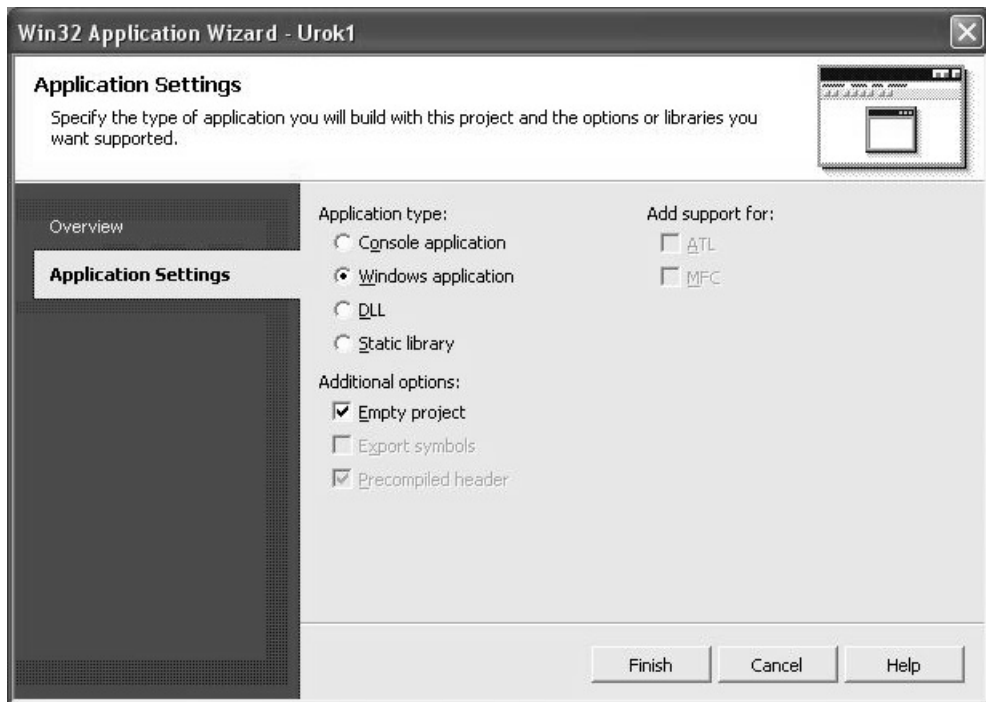


Рис. 1.1. Диалоговое окно **Win 32 Application Wizard - Urok1**

После чего нажмите кнопку **Finish**. С помощью этих операций мы создали обыкновенное пустое приложение, в котором нет ни строчки кода. С ним и будем работать. С левой стороны экрана в Visual C++. NET находится панель, содержащая три вкладки: **Solution Explorer**, **Resource View** и **Class View**. Выбрав вкладку **Solution Explorer**, вы увидите дерево проекта **Urok1**. При нажатии на знак "+" рядом с названием проекта откроется ветвь, в которой будут находиться папки: **Source File** (файлы кода имеют, как правило, расширения с и сpp) и **Header File** (заголовочные файлы, обычно с расширением h). Добавим пустой файл в папку **Source File**. В нем мы будем писать код

нашей программы. Для добавления пустого файла в папку **Source File** необходимо выполнить следующие действия.

1. Нажмите правой кнопкой мыши на папке **Source File**, в появившемся меню выберите пункт **Add**. Перейдите по стрелке в выпадающее меню, нажмите на пункте **Add New Item**. На экране появится диалоговое окно **Add New Item - Urok1**, показанное на рис. 1.2.
2. В поле **Templates** укажите пункт **C++ File (.cpp)**. При этом будет создан пустой файл, в котором и пишется код программы.
3. В поле **Name** дадим название созданному файлу `WindowsBazis`. Нажмите кнопку **Open**. После чего у вас во вкладке **Solution Explorer - Urok1** в папке **Source File** появится пустой файл `WindowsBazis.cpp`.

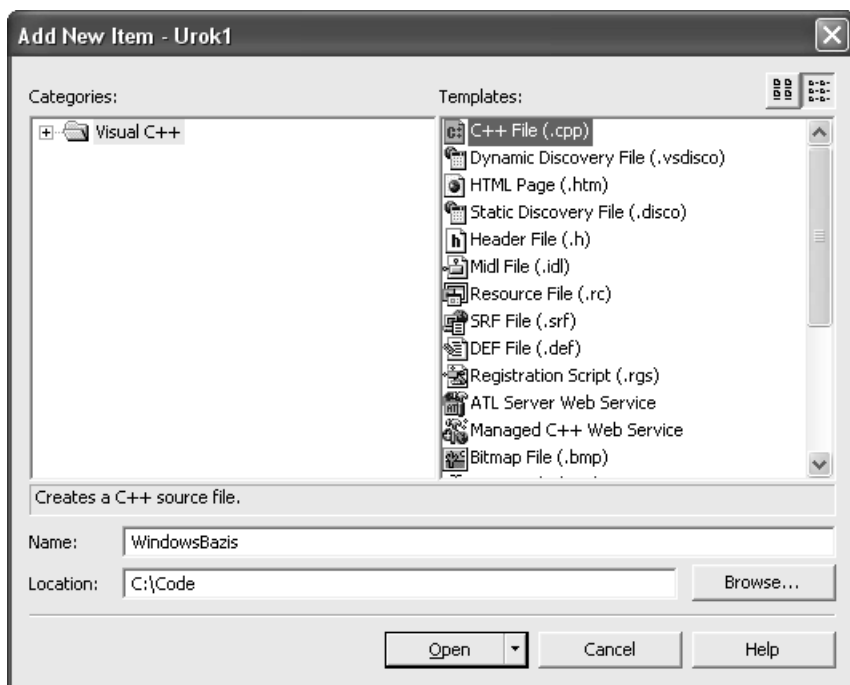


Рис. 1.2. Диалоговое окно **Add New Item - Urok1**

Общая модель сообщений в Windows

Windows очень интересная операционная система и все общение с приложениями строится на модели событий. Любое оконное приложение в момент своего существования общается с Windows посредством сообщений.

Таких сообщений может быть сколько угодно. Чтобы систематизировать их, существует т. н. очередь событий, реализованная в виде обыкновенного цикла. Каждое из поступивших сообщений ждет своей очереди для обработки, после чего изымаются операционной системой Windows. А поскольку Windows очень большая и многозадачная операционная система, то приложений в какой-то определенный промежуток времени может существовать много, а значит, и поступаемых сообщений тоже. Чтобы обработать все сообщения в Windows имеется обработчик событий для всех приложений. Схема взаимодействия приложений с сообщениями представлена на рис. 1.3.



Рис. 1.3. Модель событий Windows

Исходя из модели событий, предлагаемой Windows, получается, что нам необходимо создать окно и запустить обработчик сообщений. Отправной точкой всех приложений Windows является функция `WinMain()`, в которой происходит создание окна и обработка событий. Поэтому вначале мы создадим функцию `WinMain()`, в которой опишем класс `windowsclass`, в котором укажем размеры, стиль, цвет, фон предполагаемого окна. Потом мы создадим окно при помощи функции `CreateWindowEx()`. Далее зарегистрируем окно (печать в налоговой о том, что мы есть и готовы платить налоги), и в конце создадим обработчик событий, куда будем посылать всякие нехорошие письма.

Если в начале этого урока вы не создали проект для работы, сделайте это, и приступим к программированию в файле `WindowsBazis.cpp`.

Главная функция *WinMain*

Первое, что надо сделать, — это включить в код главный заголовочный файл `windows.h`, содержащий объявление классов, необходимых для программирования в среде Windows:

```
#include <windows.h>
```

Далее идет главная входная функция всех приложений Windows: `WinMain()`, прототип которой выглядит следующим образом:

```
int WINAPI WinMain(
    HINSTANCE    hinstance,
    HINSTANCE    hprevinstance,
    LPSTR        lpCmdLine,
    int          nCmdShow)
```

Функция `WinMain()` имеет следующие параметры:

- ❑ `hinstance` — это дескриптор создаваемого приложения, т. н. "хендл" (handle). Генерируется операционной системой Windows для дальнейшего отслеживания его использования. Можно сказать, что это своего рода метка создаваемого вами окна;
- ❑ `hprevinstance` — этот параметр уже не используется. Он был актуален при использовании Windows 3.1, оставлен просто для совместимости и всегда равен 0;
- ❑ `lpCmdLine` — указатель на командную строку, которая идет сразу за именем запускаемой команды;
- ❑ `nCmdShow` — это значение указывает, как создаваемое окно будет отображаться на экране монитора при создании. Данный параметр имеет большое количество всевозможных флагов, определяющих вид создаваемого окна. Не вижу большого смысла в разборе этих флагов, нам важнее программирование графики с помощью DirectX 9, нежели системное программирование. Но если вам это все же интересно, то можно найти нужную информацию в справочном материале Win32 API.

Разбираем класс *windowclass*

Далее нам нужно создать класс `windowclass`. *Класс* — это то, что позволяет различать вид и назначения окон. Обычно говорят, что окна отличаются друг от друга классами. Windows уже имеет большое количество заранее предопределенных классов, и все, что надо сделать, это сохранить информацию о создаваемом классе `windowclass` в структуре `WNDCLASSEX`. Давайте создадим класс `windowclass`, отвечающий нашим запросам:

```
WNDCLASSEX windowclass;    // наш класс
```

А теперь рассмотрим прототип структуры `WNDCLASSEX`, необходимый для создания класса:

```
typedef struct_WNDCLASSEX {
    UINT        cbSize;
    UINT        style;
```

```

WINDPOC      LpfnWndProc;
int          cbClsExtra;
int          cbWndExtra;
HANDLE       hInstance;
HICON        hIcon;
HCURSOR      hCursor;
HBRUSH       hbrBackground;
LPCTSTR      LpszMenuName;
LPCTSTR      LpszClassName;
HICON        hIconSm;
} WINDCLASSEX;

```

Структура имеет следующие параметры:

❑ `cbSize` — это размер создаваемой структуры. Не будем терять время и сразу при рассмотрении параметров будем заполнять поля структуры `WINDCLASSEX`: `windowclass.cbSize = sizeof(WINDCLASSEX)`. Здесь мы указываем размер создаваемой структуры;

❑ `style` — содержит множество различных флагов стилей окна, которые можно комбинировать с помощью операции побитового ИЛИ "|". Заполняем поле `style`:

```
windowclass.style = CS_VREDRAW|CS_HREDRAW|CS_OWNDC|CS_DBLCLKS,
```

комбинируя различные флаги. Рассмотрим использующиеся флаги. Если у вас появится желание узнать о них больше, то вы можете обратиться к справочной системе Win 32:

- `CS_VREDRAW` — если размер высоты окна меняется или окно было перемещено, то требуется перерисовка всего окна;
- `CS_HREDRAW` — если ширина окна меняется или окно было перемещено, то требуется перерисовка всего окна;
- `CS_OWNDC` — для каждого окна данного класса дается свой контекст устройств;
- `CS_DBLCLKS` — при совершенном в окне двойном щелчке мышью окну посылается информация о двойном щелчке;

❑ `LpfnWndProc` — это указатель на функцию обратного вызова. В любом приложении Windows существует цикл обработки событий, при работе которого и происходит вызов `MainWinProc`, поэтому необходимо записать следующее: `windowclass.LpfnWndProc = MainWinProc`;

❑ два поля, `cbClsExtra` и `cbWndExtra`, предназначены для хранения дополнительной информации. Однако ими редко кто пользуется, по крайней мере, нам в программировании DirectX они не понадобятся. Поэтому напишем: `windowclass.cbClsExtra = 0` и `windowclass.cbWndExtra = 0`;

- ❑ `hInstance` — параметр, отвечающий за экземпляр создаваемого приложения, передаваемый функцией `WinMain()`: `windowsclass.hInstance = hinstance;`
- ❑ `hIcon` — служит для определения пиктограммы вашего приложения. С помощью функции `LoadIcon()` можно загрузить собственную или системную пиктограмму. Прототип функции `LoadIcon()` выглядит следующим образом: `HICON LoadIcon(HINSTANCE hInstance, LPCTSTR LpIconName)`. Параметры прототипа такие:
 - `hInstance` — экземпляр приложения. Чтобы загрузить стандартную пиктограмму, используется значение `NULL`;
 - `LpIconName` — это идентификатор загружаемого ресурса пиктограммы. Здесь мы используем `IDI_APPLICATION` — пиктограмму по умолчанию. Вообще существует много всевозможных идентификаторов для загрузки разных пиктограмм, но сейчас задействуем этот. В дальнейшем, при программировании своей игры вы захотите создать свою пиктограмму для замены системной. На основании прототипа функции `LoadIcon()`, можно записать: `windowsclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);`
- ❑ `hCursor` — этот параметр отвечает за курсор и, по сути, практически идентичен предыдущему параметру `hIcon`, за исключением названия функции, используемой для загрузки курсора. Записывается следующим образом: `windowsclass.hCursor = LoadCursor(NULL, IDC_ARROW)`. Стандартный идентификатор стрелки `IDC_ARROW`. Также, как и в случае с пиктограммами, существует более десятка различных идентификаторов для курсора, но для игр обычно рисуются свои;
- ❑ `hbrBackgroud` — это поле структуры, отвечающей за цвет фона окна. Чтобы закрасить фон определенным цветом, нужно использовать функцию `GetStockObject()`, принимающую один параметр в виде флага, определяющего цвет кисти. Для закраски фона серым цветом запишем следующее:
`indowsclass.hbrBackgroud = (HBRUSH)GetStockObject(GRAY_BRUSH).`

Впоследствии, когда мы завершим написание кода окна, можно попробовать закрасить фон кистями других цветов, но в `DirectX` это не будет иметь никакого значения, т. к. для этих целей определены свои средства. Ниже приведены некоторые кисти различных цветов:

- `GRAY_BRUSH` — это серый цвет, которым мы закрашиваем фон окна (правильнее называть — серая кисть);
- `BLACK_BRUSH` — черная кисть;
- `WHITE_BRUSH` — белая кисть;

- LTGRAY_BRUSH — светло-серая кисть;
- DKGRAY_BRUSH — темно-серая кисть;
- HOLLOW_BRUSH — пустая кисть;

- `LpszMenuName` — поле, предназначенное для подключения стандартного меню к окну. У нас стоит значение `NULL`, потому что мы не будем использовать меню в наших проектах: `windowsclass.LpszMenuName = NULL;`
- `LpszClassName` — задается название вашего класса окна для создаваемого приложения. Могут существовать (что и происходит) сразу несколько приложений, при создании которых использовался класс `windowsclass`, а ваша операционная система должна их каким-то образом различать, именно для этого и служит это поле. Здесь вы можете дать любое название, главное потом в них не запутаться: `windowsclass.LpszClassName = "WINDOWSCLASS";`
- `hIconSm` — это дескриптор малой пиктограммы, которая выводится на панель задач Windows в полосе заголовка вашего окна. Воспользуемся стандартной пиктограммой. Для этого необходимо записать следующее: `windowsclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);`

Закончив заполнение всех полей структуры, посмотрим, что получилось:

```
WINDCLASSEX windowsclass;
windowsclass.cbSize = sizeof(WINDCLASSEX);
windowsclass.style = CS_VREDRAW|CS_HREDRAW|CS_OWNDC|CS_DBLCLKICKS;
windowsclass.LpfWndProc = WindowProc;
windowsclass.cbExtra = 0;
windowsclass.cbWndExtra = 0;
windowsclass.hInstance = hinstance;
windowsclass.LoadIcon(NULL, IDI_APPLICATION);
windowsclass.LoadCursor(NULL, IDI_ARROW);
windowsclass.hbrBackground = (HBRUSH)GetStockObject(GRAY_BRUSH);
windowsclass.LpszMenuName = NULL;
windowsclass.LpszClassName = "WINDOWSCLASS";
windowsclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
```

После того как вы заполнили и сохранили в переменной `windowsclass` все значения, класс `windowsclass` нужно зарегистрировать. Делается это с помощью функции `RegisterClassEx()`. Принимаемый параметр и есть регистрируемый класс. Класс для каждого создаваемого приложения может быть зарегистрирован только один раз:

```
RegisterClassEx(& windowsclass);    // регистрируем созданный класс
```

Создаем окно

После того как класс зарегистрирован, можно создавать окно. Это делается с помощью функции `CreateWindowEx()`. Общий вид этой функции следующий:

```
HWND CreateWindowEx(  
    DWORD        dwExStyle,  
    LPCTSTR      LpClassName,  
    LPCTSTR      LpWindowName,  
    DWORD        dwStyle,  
    int          X,  
    int          Y,  
    int          nWidth,  
    int          nHeight,  
    HWND         hWindParent,  
    HWND         hMenu,  
    HINSTANCE    hInstance,  
    LPVOID       LpParam);
```

Параметры функции `CreateWindowEx()` такие:

- ❑ `dwExStyle` — флаг стилей окна. Используется редко, в основном с флагом `WS_EX_TOPMOST`, для того, чтобы ваше окно появлялось поверх других окон. Либо просто значение `NULL`, чтобы игнорировать данный параметр;
- ❑ `LpClassName` — это имя класса создаваемого окна. В нашем случае `WINDOWSCCLASS`;
- ❑ `LpWindowName` — это заголовок окна. Здесь можно написать все, что угодно, будем использовать название нашей цели, а именно "Базовое Окно для DirectX". Это название будет меняться каждый раз по смыслу следующего урока;
- ❑ `dwStyle` — это флаг, описывающий стиль и поведение создаваемого окна. Имеется небольшое количество разных флагов, вот некоторые из них:
 - `WS_OVERLAPPED` — окно с полосой заголовка и рамкой;
 - `WS_VISIBLE` — изначально видимое окно;
 - `WS_CAPTION` — окно с полосой заголовка (включает в себя стиль `WS_BORDER`);
 - `WS_BORDER` — окно в тонкой рамке;
 - `WS_ICONIC` — окно изначально минимизировано;
 - `WS_OVERLAPPEDWINDOW` — перекрывающееся окно (включает в себя следующие шесть стилей: `WS_OVERLAPPED`, `WS_CAPTION`, `WS_SYSMENU`,

`WS_THICKFRAME`, `WS_MINIMIZEBOX` и `WS_MAXIMIZEBOX` — довольно функциональный флаг, его мы и будем использовать;

- `WS_MINIMIZE` — изначально минимизированное окно;
- `WS_MAXIMIZE` — изначально максимизированное окно;
- `WS_MAXIMIZEBOX` — окно с кнопкой `Maximize` (обязательно используется со стилем `WS_SYSMENU`);
- `WS_MINIMIZEBOX` — окно с кнопкой `Minimize` (обязательно используется со стилем `WS_SYSMENU`);

□ `x`, `y` — это позиция верхнего левого угла окна в пикселах. Можно указать координаты или воспользоваться флагом `CW_USEDEFAULT`, оставив значения по умолчанию, укажем `x`, равное 300, а `y`, равное 150, установив тем самым координаты левого верхнего угла в пикселах;

□ `nWidth` и `nHeight` — ширина и высота окна в пикселах, также можно использовать флаг `CW_USEDEFAULT` или указать необходимые размеры, например, 500 на 400 пикселей. Именно этот размер задан в листинге 1.1;

□ `hWndParent` — дескриптор родительского окна. Если родительского окна нет, используется значение `NULL`;

□ `hMenu` — дескриптор меню, т. е. если у вас есть меню, вы можете указать его дескриптор, и оно будет присоединено к вашему окну. Если меню нет, то устанавливается значение `NULL`;

□ `hInstance` — экземпляр вашего приложения. Использует значение `hinstance` функции `WinMain()`.

На данный момент нам известны все значения параметров функции. Создадим окно. Объявим дескриптор окна и вызовем функцию `CreateWindowEx()`:

```
HWND hwnd;    // дескриптор окна
if (!(hwnd = CreateWindowEx (NULL, "Windowsclass",
    "Базовое Окно для DirectX", WS_OVERLAPPEDWINDOW|WS_VISIBLE,
    0, 0, 500, 400, NULL, NULL, hinstance, NULL)))
return (0);
```

Кстати, обратите внимание на конструкцию `ifelse`, с помощью которой происходит обработка ошибок.

После того как окно создано, его нужно явно вывести на экран и обновить. Делается это двумя функциями:

```
ShowWindow(hwnd, ncmdshow);    // выводим окно
UpdateWindow(hwnd);            // обновляем окно
```

Обрабатыаем события в Windows

В Windows необходимые события обрабатываются с помощью созданного вами обработчика событий. На каждый класс можно определить свой обработчик событий, чем больше их будет, тем лучше функциональность вашей программы. В процессе работы любого приложения, в т. ч. и вашего, генерируется масса разнородных сообщений, которые передаются в т. н. очередь. События, происходящие с вашим окном, попадают в раздел очереди именно вашего окна. После чего главный обработчик событий изымает их и передает в функцию `MainWinProc()` для дальнейшей обработки. Событиями, которым не назначен обработчик, будет заниматься сама система Windows.

Разберем прототип функции `MainWinProc()`:

```
LRESULT CALLBACK MainWinProc(  
    HWND        hwnd,  
    UINT        msg,  
    WPARAM      wParam,  
    LPARAM      lParam);
```

Параметры функции `MainWinProc()` следующие:

- ❑ `hwnd` — дескриптор окна. Требуется для определения, от какого окна получено сообщение, если сразу открыто несколько окон, принадлежащих одному и тому же классу;
- ❑ `msg` — идентификатор события, которое должно быть передано в функцию `MainWinProc()` для обработки;
- ❑ `wparam` и `lparam` — являются дополнительными параметрами обрабатываемого события. Для написания обработчика событий обычно используется конструкция `switch(msg)`, в ней, на основе идентификатора сообщений `msg`, используются дополнительные параметры `wparam` и `lparam`. Рассмотрим некоторые сообщения для данных параметров, которые могут пригодиться для разработки игры. Но если вы пишете приложение Win32 без использования DirectX, тогда нужно гораздо больше информации:
 - `WM_PAINT` — данное сообщение посылается при необходимости перерисовки всего окна, которое могло быть перемещено, увеличено в размере и т. д.;
 - `WM_DESTROY` — это сообщение посылается Windows, когда окно должно быть закрыто. Можно сказать, что это такой идентификатор, который сообщает о том, что нужно освободить все захваченные приложением ресурсы;
 - `WM_QUIT` — после освобождения ресурсов вызывается данное сообщение, которое и закрывает приложение.

Объединив все вместе, мы получим функцию WinProc:

```

LRESULT CALLBACK MainWinProc(HWND hwnd, UINT msg,
                               WPARAM wparam, LPARAM lparam)
{
    switch(msg)
    case WM_PAINT:
        break;
    case WM_DESTROY:
    {
        PostQuitMessage(0);
        return 0;
    } break;
    return (DefWindowProc(hwnd, msg, wparam, lparam));
}

```

Функция `PostQuitMessage(0)` ставит в очередь сообщение `WM_OUIT`, которое и закроет ваше приложение. `WM_OUIT` применяется непосредственно в главном обработчике событий. Функция `DefWindowProc()` обрабатывает по умолчанию те сообщения, которые вы не используете.

Главный обработчик событий

Рассмотрим главный обработчик событий, применяемый в следующем примере:

```

while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg); // преобразование клавиатурного ввода
    DispatchMessage(&msg); // обработка и пересылка сообщений в WinProc
}

```

Прототип функции `GetMessage()` выглядит следующим образом:

```

BOOL    GetMessage(
LPMSG    LpMsg,
HWND     hWnd,
UINT     wMsgFiltrenMin,
UINT     wMsgFiltrenMax);

```

Параметры функции `GetMessage()` такие:

- `LpMsg` — указатель на структуру, отвечающую за сообщения;
- `hWnd` — дескриптор создаваемого окна;

- ❑ `wMsgFiltrenMin` — это первое сообщение, пришедшее на обработку;
- ❑ `wMsgFiltrenMax` — это последнее обрабатываемое сообщение.

Цикл `while` выполняется до тех пор, пока функция `GetMessage()` возвращает отличное от нуля значение. Работа самого цикла выглядит довольно просто: при поступлении сообщения из очереди оно обрабатывается функцией `TranslateMessage()` и передается функции `DispatchMessage()`, которая вызывает для обработки `MainWinProc()`, передавая ей всю необходимую информацию.

Все что получилось в результате, приведено в листинге 1.1.

Листинг 1.1. Файл `WindowsBazis.cpp`

```
//-----
//  WindowsBazis.cpp
//  данное приложение создает простейшее окно в Windows
//-----
#include <windows.h>    // подключаем заголовочный файл Windows
//-----
//  функция
//  MainWinProc()
//  здесь происходит обработка сообщений
//-----

LRESULT CALLBACK MainWinProc(HWND  hwnd,    // дескриптор окна
                              UINT   msg,    // идентификатор сообщения
                              WPARAM wParam, // дополнительная информация
                              LPARAM lParam) // дополнительная информация
{
    switch(msg)
    {
        case WM_PAINT:
            break;
        case WM_DESTROY:
            {
                PostQuitMessage(0);
                return(0);
            } break;
    }
}
```



```
    NULL,                                // дескриптор родительского окна
    NULL,                                // дескриптор меню
    hinstance,                           // дескриптор приложения
    NULL))                               // указатель на данные окна
return(0);
ShowWindow(hwnd, SW_SHOWDEFAULT);      // нарисуем окно
UpdateWindow(hwnd);                     // обновим окно
while(GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
return(msg.wParam);
}
```

Окончательный вариант листинга 1.1 также можно найти на прилагаемом компакт-диске в папке Code\Urok1\WindowsBasis.cpp.



Рис. 1.4. Базовое окно для DirectX

Немного времени стоит уделить компиляции исходного кода. Сейчас вы находитесь в окне, где у вас открыт проект Urok1. Выделите или просто откройте файл WindowsBazis.cpp.

Нажмите клавишу <Ctrl> и, не отпуская ее, клавишу <F7>; произойдет компиляция исходного файла WindowsBazis.cpp. После этого нажмите клавишу <F7> — произойдет линковка. Для просмотра результата воспользуйтесь комбинацией клавиш <Ctrl>+<F5>. Пример окна представлен на рис. 1.4.

Итоги урока

На этом уроке было создано простое оконное приложение, в которое на протяжении всей книги мы будем интегрировать свой код и функции из библиотеки DirectX 9. Как выяснилось, оконное приложение может создаваться с любыми необходимыми для вас свойствами. На *уроке 2* вас ждет большой вводный курс в DirectX 9. Будет раскрыта концепция построения сцены в DirectX3D, а также будет затронута технология COM, на платформе которой базируется вся библиотека DirectX.

Урок 2



DirectX 9

На этом уроке мы познакомимся с библиотекой DirectX 9 SDK, объединяющей в себе ряд компонентов для работы с графикой, звуком, видео, устройствами ввода, а также другими мультимедийными компонентами. В конце урока установим DirectX 9 SDK с компакт-диска, прилагаемого к книге, на ваш компьютер.

Что такое DirectX? Представьте себе ситуацию: вы разрабатываете игру и как разработчик (не говоря уже об издателе) желаете, чтобы игра могла работать на как можно большем количестве компьютеров. Но для этого вам необходимо учесть все мыслимые и немыслимые видео-, звуковые и сетевые карты, рули, мыши, клавиатуры и еще целый "ворох" разного оборудования. Гораздо легче создать определенный стандарт, благодаря которому написание кода, например, для устройств ввода, можно сделать единожды, а работать он будет всегда и везде. К этому и стремится корпорация Microsoft уже много лет, и с каждой новой версией DirectX ей это удается все лучше и лучше.

DirectX — это мультимедийная библиотека, позволяющая напрямую работать с аппаратным обеспечением компьютера в обход традиционных средств платформы Win32. Тесное взаимодействие DirectX и оборудования с драйверами, написанными производителем данного оборудования, дают возможность полностью отвлечься от аппаратной части и сосредоточить свое внимание на создании правильного кода. Вся DirectX 9 делится на компоненты, отвечающие за ту или иную часть работы библиотеки:

- ❑ DirectX Graphics;
- ❑ DirectInput;
- ❑ DirectMusic;
- ❑ DirectSound;
- ❑ DirectPlay;
- ❑ DirectShow;
- ❑ DirectSetup.

На рис. 2.1 представлена общая схема взаимодействия DirectX 9 с аппаратным обеспечением компьютера.

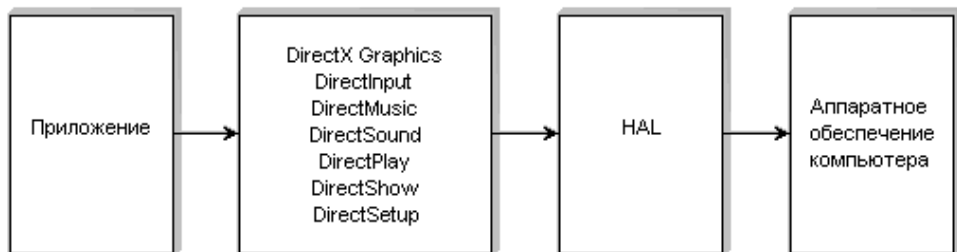


Рис. 2.1. Схема взаимодействия DirectX 9

Компоненты DirectX 9

В этой книге мы познакомимся с четырьмя составляющими DirectX 9 — это Direct3D, входящий в состав DirectX Graphics, DirectInput, DirectMusic и DirectSound. Все перечисленные компоненты необходимы в программировании компьютерных игр. Что касается звука, то вообще достаточно будет DirectMusic, но в целях общего ознакомления рассмотрим некоторые составляющие из DirectSound.

DirectX Graphics в ранних версиях (до седьмой включительно) был разделен на две части: DirectDraw и Direct3D. DirectDraw отвечал за двухмерную графику, тогда как Direct3D за трехмерную. В восьмой версии DirectX DirectDraw был объединен в Direct3D. Произошло это по понятным причинам, настала эра трехмерных игр и удивить кого-то двухмерной графикой уже было невозможно. С выходом DirectX 9 почти ничего не изменилось, хотя конечно возможность к доступу DirectDraw по-прежнему имеется, но вот насколько это актуально? В любом случае мы будем рассматривать только Direct3D, отвечающий за работу с трехмерной графикой.

DirectInput — это компонент, отвечающий за работу со всеми устройствами ввода: клавиатурой, мышью, джойстиком, рулем, шлемом "виртуальной реальности", с устройствами виброотдачи и обратной связи. Данный компонент обеспечивает работу устройств и нужную скорость доступа к приложению за счет прямого взаимодействия программы с аппаратным обеспечением компьютера.

DirectMusic, — как видно из названия, отвечает за работу с музыкой, и в большей степени его задачей является работа с MIDI (Musical Instrument

Digital Interface), но есть возможность проигрывания WAV-файлов. Это очень сложный компонент и одним проигрыванием MIDI- и WAV-файлов он не ограничивается.

DirectSound — эта часть DirectX 9 отвечает за звук и позволяет применять основные, вспомогательные, статистические и потоковые буферы, обеспечивающие всевозможные эффекты звучания. Также может воспроизводить WAV-файлы.

DirectPlay призван обеспечить ваше приложение взаимодействием с локальной сетью или сетью Internet для многопользовательского режима игр. Построен на своем протоколе, обеспечивающем работу со всеми протоколами сети.

DirectShow необходим для работы с потоковым видео, в частности, служит для захвата и воспроизведения видео различных форматов: MPEG (Motion Picture Experts Group), MPEG Audio Layer-3 (всем известный как формат MP3) и, конечно Audio-Video Interleaved (AVI).

DirectSetup — компонент, отвечающий за установку файлов DirectX на компьютер пользователя из вашего приложения. Вы, наверно, сталкивались с ситуацией, когда до или после установки игры на экране появлялось диалоговое окно с предложением установить, например, более позднюю версию DirectX. Как раз за установку и отвечает этот компонент. Также в его обязанности входит автоматический запуск диска с игрой из дисковода.

В итоге, можно сказать, что DirectX 9 — это очень большая библиотека, отвечающая за работу с мультимедийными составляющими компьютера, обеспечивая высокую производительность, и, что самое главное, дает возможность не задумываться об аппаратной части компьютера при написании кода.

HAL

DirectX включает в себя уровень абстракции — *HAL* (Hardware Abstraction Layer — уровень абстракции оборудования), позволяющий настроить работу при любом аппаратном обеспечении.

С помощью уровня абстракции оборудования HAL происходит взаимодействие приложения с оборудованием компьютера, вне зависимости от изготовителя этого оборудования. Это дает возможность единожды написанному коду работать на любом аппаратном обеспечении.

COM

Вся библиотека DirectX построена на основе *COM* (Component Object Model) — модели составных компонентов. Самое главное, это то, что вам не

придется углубляться в сущность СОМ-технологии, вся работа с СОМ основана на вызовах соответствующих функций. Однако понимать общие принципы СОМ необходимо, поэтому немного теории не помешает.

СОМ — это своего рода спецификация, не зависящая от платформы и языка программирования, определяющая стандарт по использованию и созданию программных компонентов. У вас дома есть несколько розеток и несколько вилок для 220 V, любую вилку вы можете включить в любую розетку — это и есть стандарт, сопровождающийся своей спецификацией. Спецификация СОМ устанавливает определенный стандарт для разрабатываемой компонентной архитектуры. Программные компоненты состоят из исполняемого кода, представленного в виде динамических библиотек (DLL) и EXE-файлов платформы Windows.

В составе СОМ имеется API, называемая СОМ-библиотекой; с ее помощью достигается управление всеми компонентами. Каждый из программных компонентов реализует определенное количество СОМ-объектов, доступ к которым осуществляется посредством интерфейсов, которые, в свою очередь, состоят из функций. С помощью этих функций и происходит взаимодействие с СОМ-объектом. Интерфейсы в СОМ — это строго определенная структура, содержащая массив указателей на функции. Каждый из компонентов DirectX, например Direct3D, включает ряд интерфейсов с набором функций, с помощью которых мы будем получать доступ к СОМ-объекту. Все интерфейсы в DirectX имеют свое обозначение и начинаются с английской заглавной буквы "I". Далее указывается название СОМ-объекта, например Direct3D, затем номер версии используемой библиотеки DirectX, например: IDirect3D9. Все интерфейсы строго определены и не могут изменяться, что позволяет иметь доступ к более старым версиям интерфейсов.

СОМ-объекты и интерфейсы идентифицируются в системе 128-битным глобально-уникальным идентификатором (GUID), или, как говорят, "гидом", назначаемым при разработке компонента. Аббревиатура для записи идентификатора интерфейса существует в виде IID (Interface Identifier — идентификатор интерфейса), а для СОМ-объекта — CLSID (Class Identifier — идентификатор класса).

Интерфейс *IUnknown*

Все СОМ-интерфейсы наследуются от интерфейса *IUnknown*, причем дословный перевод означает "неизвестный". Забавно, не правда ли? Интерфейс *IUnknown* содержит три функции: `AddRef()`, `Release()` и `QueryInterface()`, а поскольку все интерфейсы наследуются от *IUnknown*, то все три функции имеются в каждом из СОМ-интерфейсов.

СОМ-объект содержит счетчик, определяющий количество используемых интерфейсов. С каждым новым задействованным интерфейсом счетчик увеличивается на единицу. С помощью функции `AddRef()` происходит увеличение счетчика определенного СОМ-объекта. Пользоваться нам этой функцией не придется, скажем так, увеличение счетчика ссылок происходит автоматически.

После того как интерфейс отработал, он освобождается, и внутренний счетчик СОМ-объекта уменьшается на единицу. Как только счетчик обнуляется, СОМ-объект выгружает себя из памяти, и в тот же момент все созданные им объекты будут выгружены системой. За уменьшение счетчика ссылок на единицу отвечает функция `Release()`. А вот этой функцией мы будем пользоваться постоянно. Операция, выполняемая функцией, еще называется "освобождением захваченных ресурсов". Далее на уроках мы создадим специальную функцию, которая будет освобождать захваченные ресурсы путем вызова функции `Release()`.

Как уже говорилось, в DirectX 9 нет интерфейса `DirectDraw`, отвечающего за двумерную графику, но вдруг по какой бы то ни было причине, вам захотелось получить доступ к этому интерфейсу. Например, к седьмой версии `DirectDraw7`. Поскольку нам известно, что реализация интерфейсов строго определена для совместимости с более ранними версиями, то такая возможность у вас, конечно, имеется. Для этого существует функция `QueryInterface()`. Если рассмотреть прототип этой функции, то все сразу станет ясно:

```
HRESULT QueryInterface(  
    REFIID iid,  
    void** ppvDest)
```

Функция `QueryInterface()` имеет следующие параметры:

- `iid` — это идентификатор запрашиваемого интерфейса (в соответствии с аббревиатурой IID);
- `ppvDest` — адрес указателя на полученный интерфейс.

Тогда, чтобы получить доступ к интерфейсу `DirectDraw7`, необходимо написать такой код:

```
Указатель_на_интерфейс -> QueryInterface(IID_IDirectDraw7,  
                                           &Указатель_нужной_версии);
```

Здесь, конечно, не хватает нескольких операций, но все станет на свои места по мере прочтения этой книги. При успешном выполнении данной функции вам станет доступен интерфейс `DirectDraw7`.

Direct3D 9

Direct3D — это самый мощный и самый главный компонент DirectX 9, отвечающий за отрисовку трехмерной графики, наложения на объект текстур, за вершинные и пиксельные шейдеры и многое другое. С помощью Direct3D существует возможность встраивать трехмерную графику в приложение для платформы Windows. Имея в своем распоряжении большой набор различных функций, структур, макросов, флагов, разработчик практически отодвинут от аппаратной части компьютера и сосредоточен на программном коде. Сам же программный код, а точнее множество функций трехмерной графики, очень легко встраиваются и взаимодействуют с приложением.

Самой главной составляющей Direct3D является процесс рендеринга (rendering). *Рендеринг* — это визуализация трехмерного объекта на экране монитора. Здесь важно понимать, что любой объект хранится в видео- или системной памяти компьютера, как разобранный на части конструктор, где каждая из частей отвечает за ту или иную составляющую объекта. Например, робот из конструктора изначально состоит из головы, ног, рук, туловища, точно так же и объект разбит на части, состоящие из текстур, света и материала, полигонов и многого другого. Чтобы увидеть целого робота в конструкторе, его нужно собрать из частей, чтобы увидеть объект на экране монитора, он должен пройти через процесс рендеринга, который произведет сборку объекта. Конечно, на самом деле я здесь немного упрощаю общую концепцию, также под объектом подразумевается не только какая-то отдельно взятая трехмерная модель, но и весь окружающий ее мир. В Direct3D это носит название *сцены*.

Весь процесс рендеринга состоит из нескольких механизмов — тесселяции, трансформации, освещения и растеризации. С помощью этих механизмов или, как говорят, *конвейера рендеринга*, достигается визуализация всей сцены на экране.

Тесселяция (tessellation) — это механизм разбиения поверхности на полигоны. Полигон — это определенная площадь в трехмерном пространстве, ограниченная углами. В Direct3D полигоном обычно является треугольник.

Трансформация (transformation) — это преобразование координат объекта с помощью матриц, производящих вращение, перенос, масштабирование и другие различные матричные преобразования объекта.

Освещение (lighting) — прямая аналогия нашему миру, где все предметы имеют цвет, который в свою очередь зависит от источников освещения.

Растеризация (rasterizer) — это самый сложный механизм, который, по сути, и рисует всю сцену на экране монитора, представляя ее на пиксельном уровне, осуществляя текстурирование, Z-буферизацию, затенение, альфа-смешивание, сокрытие невидимых поверхностей и многое другое.

Построение сцены в Direct3D

Прежде чем мы начнем программирование графики средствами Direct3D, хотелось бы в общих чертах коснуться принципа построения сцены, который условно можно разбить на несколько частей.

1. *Создание оконного приложения.* Это то, чем мы занимались на первом уроке. Создав окно, в любом выбранном вами режиме и виде, вы создаете себе рабочую поверхность, куда будете последовательно интегрировать свой код и функции DirectX.
2. *Инициализация Direct3D.* Здесь вы создаете ряд указателей на необходимые вам интерфейсы, инициализируется Direct3D 9, задаются параметры представления и создается устройство Direct3D. Устройство Direct3D — это, если можно так сказать, аппаратная и программная часть вашего компьютера. То есть вы задаете разрешение экрана, частоту смены кадров, узнаете информацию о видеокарте, устанавливаете, как видеоадаптер будет обрабатывать предложенную ему информацию, задаете формат заднего буфера, устанавливаете параметры буфера глубины, вообще все то, что касается настройки аппаратной и программной частей вашего компьютера.
3. *Создание объекта.* Каждый объект, каким бы он большим не был, состоит из полигонов. Обычно это треугольники. Любой треугольник состоит из трех углов или точек. В Direct3D оперируют понятием *вершины*. Вершина задается координатами в пространстве по трем осям X , Y и Z . Зная все это, а также имея представление о тригонометрии, можно построить практически любой объект. Задавая формат вершин в зависимости от того, используете ли вы матричные преобразования для рендеринга объекта или пользуетесь преобразованным форматом вершин, для представления 2D-объекта на экране, вы тем самым создаете объект.
4. *Освещение, материал и текстура.* Реально сцену можно представить на экране, если использовать освещение и материал. Создав различные источники света, например, лампочку, солнце, огонь, и задав материал для объекта, определяющий отражение источника света, вы добавите реализма рисуемой сцене. Ну, а наложив текстуру, например, в виде паркета на пол, вы добьетесь абсолютной идентичности нашему миру.
5. *Рендеринг объекта.* Можно прорисовать сцену на экране, задавая различные значения для всех компонентов конвейера рендеринга.

Конечно, все эти этапы для наглядности разделены на условные составляющие, но общий принцип, я думаю, вам теперь понятен. Сейчас давайте рассмотрим, из каких интерфейсов состоит Direct3D 9 и как создать указатель на необходимый интерфейс. А в конце урока установим DirectX 9 SDK на ваш компьютер.

Интерфейсы Direct3D 9

Direct3D 9 содержит самое большое количество различных интерфейсов отвечающих за текстуры, вершинные и пиксельные шейдеры, буфер вершин, индексный буфер, интерфейс устройства Direct3D 9 и т. д. Все интерфейсы мы рассмотреть не в силах, поэтому разберем только непосредственно встречающиеся в книге:

- IDirect3D9;
- IDirect3DDevice9;
- IDirect3DVertexBuffer9;
- IDirect3DIndexBuffer9;
- IDirect3DTexture9;
- IDirect3DVertexShader9.

IDirect3D9 — самый главный интерфейс, из него наследуются все остальные интерфейсы. Это первый объект, который вы должны создать, и только потом можно получить доступ ко всем остальным интерфейсам и функциям. Объект интерфейса создается при помощи функции `Direct3DCreate9()`.

IDirect3DDevice9 — после того как создан объект основного интерфейса IDirect3D9, создается интерфейс устройства Direct3D. Как я уже говорил, это, по сути, определенный набор необходимых настроек аппаратной и программной составляющих компьютера, создающих устройство Direct3D, на основе которого рисуется вся сцена. Если мы говорим о графике, то очевидно, что устройство Direct3D будет представлять видеоадаптер. Этот интерфейс всегда создается вторым, с помощью функции `IDirect3D9::CreateDevice`. И, конечно же, данный интерфейс содержит еще ряд функций, осуществляющих операции с устройством Direct3D, так же это в полной мере касается и других интерфейсов.

IDirect3DVertexBuffer9 — интерфейс, с помощью которого создается буфер вершин. Каждый объект состоит из треугольников, заданных точками в пространстве (вершин). Понятно, что весь объект состоит из определенного набора вершин, и для того, чтобы все эти вершины было удобно хранить, создается буфер вершин, являющийся в общем виде массивом данных. Буфер вершин создается при помощи функции `IDirect3DVertexBuffer9::CreateVertexBuffer`.

IDirect3DIndexBuffer9 — используя этот интерфейс, вы можете создать индексный буфер, позволяющий проиндексировать большое количество вершин объекта. Например, вы имеете прямоугольник, состоящий из двух треугольников, где по крайней мере четыре вершины (по две на каждый из двух углов) имеют одинаковые координаты, т. е. повторяются. С помощью индексного буфера можно проиндексировать вершины и избежать повторения. Может быть, конечно, для треугольника это и не актуально, но для куба индексация

вершин придется к стати. При необходимости индексный буфер можно создать с помощью функции `IDirect3DIndexBuffer9::CreateIndexBuffer`.

`IDirect3DTexture9` — как видно из названия, этот интерфейс отвечает за текстуры. В его состав входит много функций, с помощью которых вы можете накладывать текстуры на объект, одну на другую и т. д.

`IDirect3DVertexShader9` — этот интерфейс, как мне кажется, отвечает за будущее игровой индустрии, по крайней мере, на несколько ближайших лет точно. Дело в том, что механизм трансформации и освещения в более серьезных приложениях потихоньку сдает свои позиции, поскольку имеет свои ограничения в использовании, и ему на смену пришли вершинные шейдеры, улучшающие качество графики. Вершинный шейдер — это программа, написанная на языке, подобном ассемблеру, и интегрированная в создаваемое приложение, позволяющая в реальном времени достичь более качественного уровня графики.

Как уже упоминалось, это далеко не все интерфейсы, а только те, с которыми нам предстоит столкнуться. В состав DirectX 9 еще входит так называемая библиотека утилит DirectX 9. Используя библиотеку, можно загружать объекты, созданные в 3DS MAX, или осуществлять матричные преобразования с объектом. Состав этой библиотеки тоже содержит множество различных интерфейсов, с некоторыми из которых мы познакомимся в этой книге.

Создание указателя на интерфейс

Чтобы создать указатель на интерфейс, нужно просто объявить переменную определенного интерфейса, в которой будет храниться указатель. Существующий, по крайней мере, два способа создания указателя на интерфейс.

Первый способ:

```
IDirect3D9* pDirect3D;  
IDirect3DDevice9* pDirect3DDevice;
```

Второй способ:

```
LPDIRECT3D9 pDirect3D;  
LPDIRECT3DDEVICE pDirect3DDevice;
```

Оба способа приводят к одному результату. Чтобы было понятно, почему это возможно, приведу спецификацию для интерфейса `IDirect3D9`, которая является показательной и для всех остальных интерфейсов:

```
typedef struct IDirect3D9 *LPDIRECT3D9 *PDIRECT3D9
```

В книге везде используется второй способ. После того как был создан указатель на интерфейс, можно попытаться создать объект. Раз уж мы создали

два указателя на интерфейсы `IDirect3D9` и `IDirect3DDevice9`, давайте дойдем до логического конца и создадим два объекта для обоих интерфейсов:

```
// указатель на интерфейс IDirect3D9
pDirect3D = Direct3DCreate9(D3D_SDK_VERSION);

// указатель на интерфейс IDirect3DDevice9
pDirect3D -> CreateDevice(D3DADAPTER_DEFAULT,
                        D3DDEVTYPE_HAL, hwnd,
                        D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                        &Direct3DParametr, &pDirect3DDevice);
```

Более подробно эти шаги будут разбираться на *уроке 3*, сейчас важно понять общий смысл всех действий, чтобы не напоминать об этом далее. В начале создается указатель на интерфейс, а точнее, переменная, в которой будет храниться указатель на интерфейс. После чего создается объект интерфейса, и указатель на этот созданный объект помещается в переменную, которая получает название — указатель на интерфейс. Поэтому всегда, когда будет упоминаться указатель на интерфейс, в сущности, это переменная, в которой хранится указатель на объект интерфейса.

А теперь давайте установим библиотеку `DirectX 9 SDK`, находящуюся на прилагаемом к книге компакт-диске, на ваш компьютер. Если вы уже это сделали, проверьте, все ли было сделано правильно. Ничего сложного нет, но пара-тройка нюансов имеется.

Установка DirectX 9 SDK

На прилагаемом к книге компакт-диске в папке `DirectX 9 SDK` лежит файл `dx90bsdk.exe` размером 218 Мбайт. На сегодняшний день это самая последняя версия продукта `DirectX 9.0b`, включающая в себя библиотеки для трех языков программирования: `C++`, `C#`, `Visual Basic`.

Установка выполняется следующим образом:

1. Дважды щелкните левой кнопкой мыши на значке файла `dx90bsdk` в папке `DirectX 9 SDK`. Это приведет к распаковке всего SDK на ваш компьютер.
2. После распаковки на экране монитора появится окно **Microsoft DirectX 9.0 SDK - InstallShield Wizard** (рис. 2.2). У вас появится возможность выбора необходимых компонентов. В пункте **DirectX Samples and Source Code** предлагаются к выбору документация, примеры, коды к трем языкам программирования, нас интересует непосредственно `C++`, все остальные языки на ваше усмотрение.

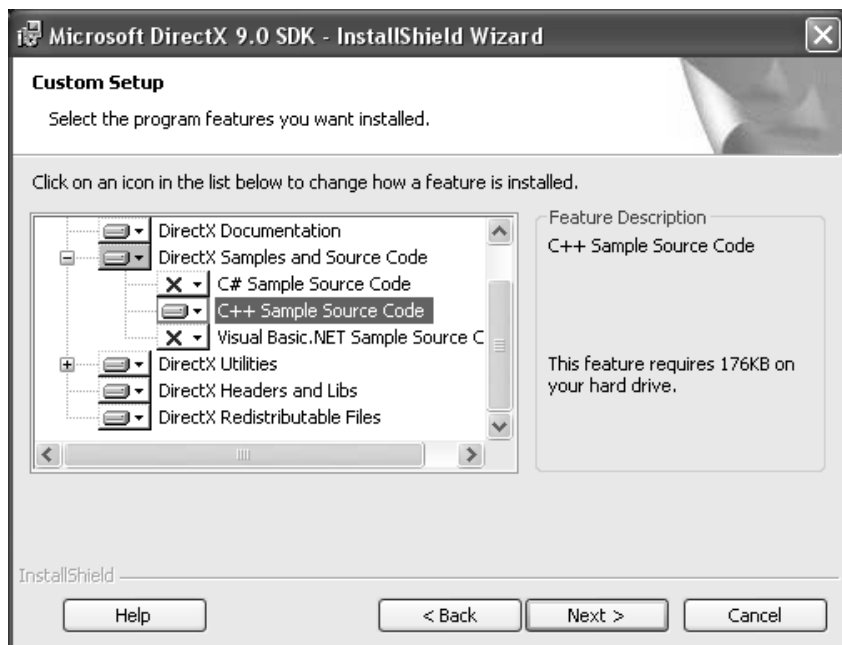


Рис. 2.2. Окно Microsoft DirectX 9.0 SDK - InstallShield Wizard

3. В диалоговом окне инсталлятора InstallShield Wizard в поле **Install to** укажите каталог для инсталляции DirectX 9, обычно это корневой каталог C:\DXSDK.
4. Нажмите кнопку **Next**, откроется диалоговое окно (рис. 2.3), где вам предлагается тип установки.
5. Установите переключатель **Debug** в области **DirectX Runtime Support**. Нажмите кнопку **Install Now** для продолжения инсталляции.
6. По окончании установки вас попросят перезагрузить компьютер, и на этом установка DirectX 9 SDK закончена, можно приступить к программированию.

В самом Visual C++ .NET при создании нового проекта к стандартным мастерам (AppWizard) добавится еще один мастер — DirectX 9 Visual C++ Wizard, с его помощью можно создать минимальное приложение с использованием DirectX 9. С этим своего рода "скелетом" можно работать, хотя лучше создать свой "костяк", удовлетворяющий вашим запросам, и добавить его в список мастеров. Для практики давайте создадим приложение с помощью мастера, откомпилируем его и посмотрим, что получится.



Рис. 2.3. Выбор типа установки

Откройте Visual C++ .NET. Выполните команду меню **File | New | Project**, появится диалоговое окно **New Project**. В поле **Templates** выделите, с помощью одного щелчка левой кнопкой мыши, строку **DirectX 9 C++ AppWizard**, в поле **Name** задайте имя проекта, например, *Demo*. В поле **Location** укажите, где будет храниться ваш проект, например, *C:\Code*. Нажмите кнопку **OK**. На экране появится окно **DirectX 9 C++ AppWizard - Demo** (рис. 2.4).

В активной вкладке **Overview** будут перечислены компоненты, которые можно установить в приложении. Ниже на доступных трех вкладках: **Project Settings**, **Direct3D Options**, **DirectInput Options** можно подключить или отключить ряд компонентов, перечисленных на вкладке **Overview**. Оставим все, как есть и нажмем кнопку **Finish**, создав тем самым проект, сформированный полностью мастером DirectX.

В Visual C++ .NET на вкладке **Solution Explorer** слева появятся два с половиной десятка файлов с разным расширением, из любопытства можете просмотреть их, но, думаю, понятного там будет мало, иначе зачем вы читаете эту книгу. Теперь откомпилируйте этот проект и посмотрите, что получилось.

На экране должно появиться простое оконное приложение, внутри него на синем фоне будет красоваться великолепный красный чайник, который с помощью клавиш **<Up>**, **<Left>**, **<Right>**, **<Down>** можно пристально рассмотреть со всех сторон.

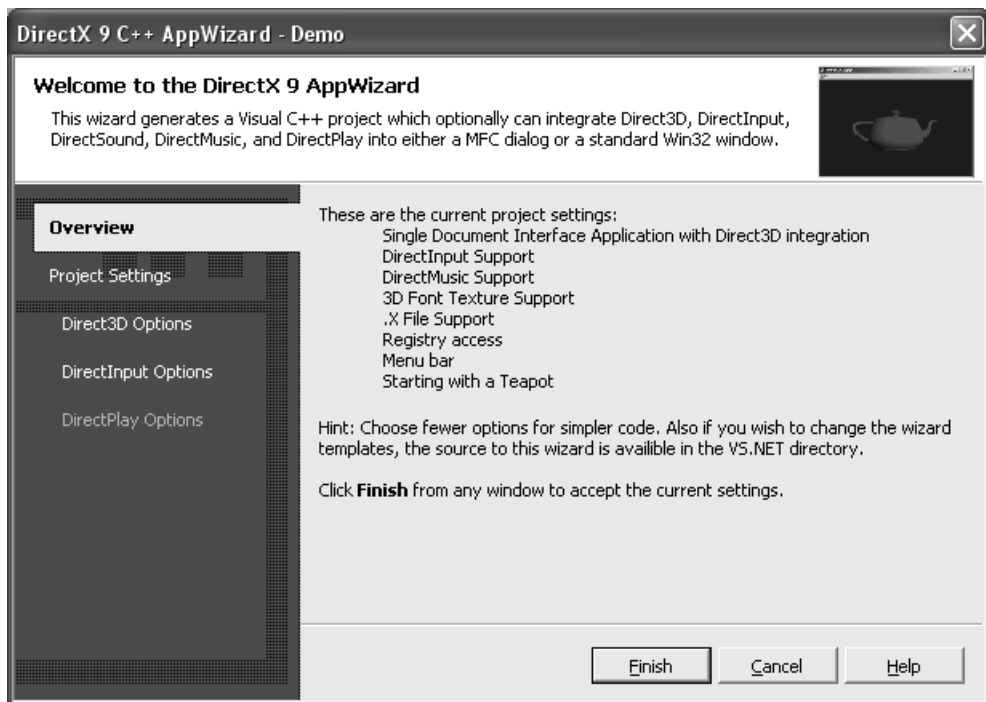


Рис. 2.4. Окно **DirectX 9 C++ AppWizard - Demo**

Итоги урока

На этом уроке была подробно рассмотрена мультимедийная библиотека DirectX 9, являющаяся стандартом в программировании игр для Windows. Мы поговорили о COM-модели, узнали, чем отличаются друг от друга COM-объекты и COM-интерфейсы, и как происходит взаимодействие между ними. Был разобран принцип построения сцены в DirectX 3D, а также созданы указатели на два основных интерфейса DirectX 3D. В конце урока на ваш компьютер был проинсталлирован пакет DirectX 9 SDK и теперь все готово для того, чтобы освоить программирование графики, чем мы и займемся на *уроке 3*.

Урок 3



Инициализация Direct3D

На *уроке 1* мы создали простое оконное приложение, появляющееся на экране монитора в указанном месте с заданным размером и закрашенным в серый цвет фоном. Очевидно, что мы можем менять размер, стиль, фон и точку вывода окна, подставляя различные значения в созданном классе `mainwindowclass`.

На *уроке 2* вы получили полную информацию о компонентах DirectX 9, также в общих чертах рассмотрели Direct3D и теперь имеете представление, с чем вам придется столкнуться.

Почти все уроки построены на примерах, т. е. имеются примеры, реализующие определенные этапы в работе. Например, на этом уроке приводится инициализация Direct3D 9, и в течение всего урока идет подробнейший разбор предлагаемого кода, а в конце рассматривается весь итоговый листинг (листинг 3.1). Более лучшего способа изъяснения на данном этапе придумать невозможно. Почему, сейчас объясню на примере все той же инициализации Direct3D 9. Существуют определенные шаги, которые вы, согласно документации по Direct3D 9 SDK, обязаны пройти, иначе у вас просто ничего не будет работать. Вам нужно создать указатели на интерфейсы, создать объекты интерфейса, установить параметры представления, узнать информацию об адаптере и т. д. Все это стандартные, можно даже сказать, жестко определенные шаги, которые необходимо пройти в заданной последовательности. Другое дело, что, конечно, можно задавать различные значения для используемых параметров функций, настраивать Direct3D 9 как вам захочется, но все же общий многоступенчатый способ работы вполне очевиден, поэтому и был избран такой способ подачи материала. По моему мнению, он приносит намного больше пользы. Каждый последующий урок основан на модификации предыдущего урока, и с каждым новым последовательным шагом вы будете узнавать все больше и больше информации. А теперь вернемся к нашему уроку.

В данный момент у нас есть оконное приложение, и на этом уроке мы займемся инициализацией Direct3D 9, подготовив тем самым базу для дальнейших уроков.

Весь урок можно разбить на следующие три этапа:

- инициализация Direct3D;
- очистка заднего буфера;
- освобождение ресурсов, захваченных Direct3D 9.

Создаем функцию для инициализации Direct3D

Для начала создадим функцию, в которой будет находиться код инициализации Direct3D 9, назовем ее `InitialDirect3D()`. Можете назвать ее как угодно, но я все же настоятельно рекомендую пока воспользоваться моими названиями, ибо вы рискуете далее запутаться в большом количестве написанного кода. В последствии, когда вы во всем разберетесь, а я на это искренне надеюсь, вы сможете сами решать, как называть созданные функции. Единственное, что хочу посоветовать, — старайтесь давать более подходящие по смыслу имена.

Начнем, но перед этим создадим новый проект. Запустите Visual C++ .NET, создайте проект, назвав его `Urok3`.

Примечание

Точно так же, как и на уроке 1, у вас должен получиться пустой проект, не забудьте в диалоговом окне **Win 32 Application Wizard - Urok3**, при создании проекта к этому уроку, перейти на вкладку **Application Settings** и установить в области **Additional options** флажок **Empty project**. Соответственно, в поле **Application type** необходимо выбрать переключатель **Windows application**.

Добавьте в проект пустой файл для программирования на C++, назвав его `InitialDirect3D.cpp`. Затем откройте файл `WindowsBazis.cpp` (см. компакт-диск, папка `Code\Urok1\WindowsBazis.cpp`) и скопируйте весь код в буфер обмена, после чего вставьте в `InitialDirect3D.cpp`. Я думаю, здесь все понятно — мы просто используем код, написанный на уроке 1.

В том месте, где мы подключаем макрос и заголовочный файл `Windows`, добавим строку кода, подключая тем самым заголовочный файл из библиотеки `DirectX 9`:

```
#include <d3d9.h>
```

Нам понадобится этот заголовочный файл, в котором объявлено большое количество классов для Direct3D 9, и, кроме того, еще необходимо подключить библиотеку `d3d9.lib`. Имеются два способа подключения библиотек в проект. Рассмотрим первый способ.

С левой стороны рабочего окна Visual C++ .NET, на вкладке **Solution Explorer**, щелкните правой кнопкой мыши на проекте **Urok3**, появится меню. В нем выделите поле **Properties**, после чего откроется диалоговое окно **Urok3 Property Pages** (рис. 3.1).

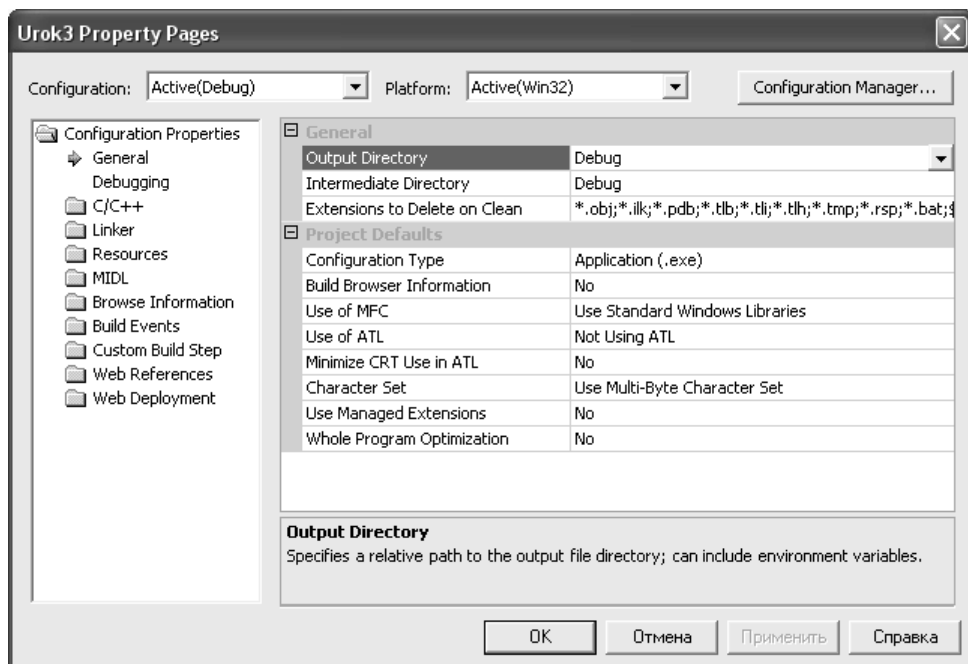


Рис. 3.1. Диалоговое окно **Urok3 Property Pages**

В левой стороне окна находится ряд папок, раскройте папку **Linker** и перейдите в папку **Input**, где в поле **Additional Dependencies** необходимо записать **d3d9.lib**. Нажмите кнопку **OK**. Таким образом, библиотека будет добавлена в ваш проект. Добавление библиотеки показано на рис. 3.2.

Все последующие библиотеки, например, **dinput.lib**, для работы с устройствами ввода, прописываются там же, добавляются они через пробел. Порядок написания значения не имеет.

Второй способ подключения библиотеки заключается в явном подключении требуемой библиотеки непосредственно в файл кода с помощью директивы препроцессора `pragma`:

```
#pragma comment(lib, "d3d9.lib")
```

Оба способа хорошо работают, выбирайте, какой больше нравится, но в книге используется первый способ подключения библиотеки.

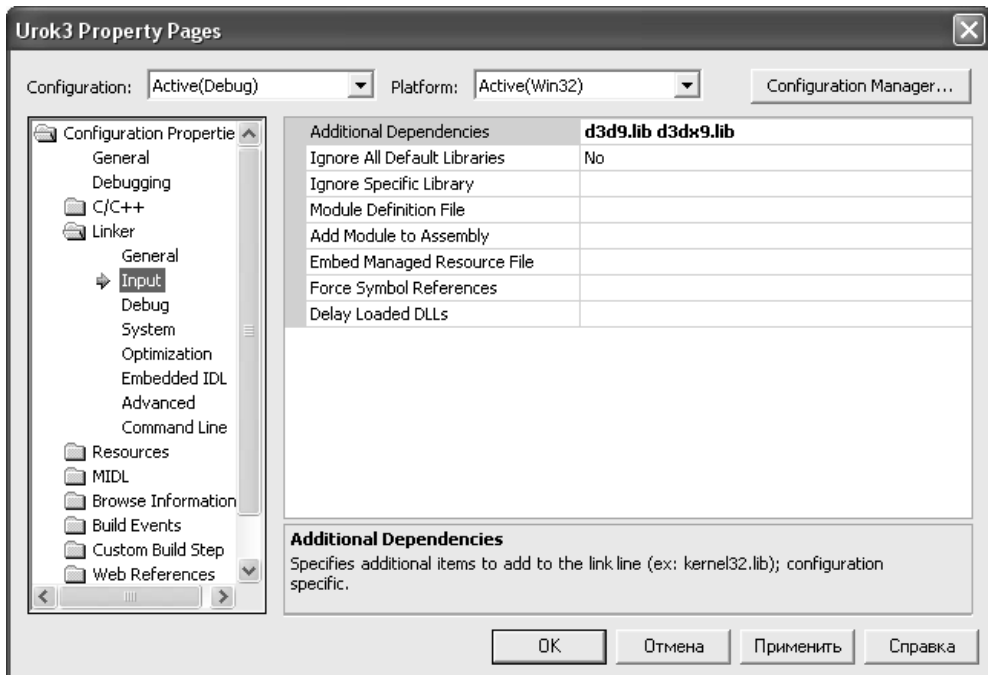


Рис. 3.2. Добавление библиотеки

Теперь давайте создадим нужные нам указатели на интерфейсы, определив их как глобальные переменные. Сразу после подключенных заголовочных файлов напишем:

```
LPDIRECT3D9 pDirect3D = NULL;
```

Это указатель на главный интерфейс IDirect3D9 для работы с трехмерной графикой. Мы создаем переменную, в которой будем хранить указатель на объект интерфейса и, как уже договорились, будем называть его указатель на интерфейс. Всем создаваемым указателям на интерфейсы необходимо присваивать значение NULL. Этим вы просто обнуляете указатель:

```
LPDDIRECT3DDEVICE9 pDirect3DDevice = NULL;
```

Так создается указатель на интерфейс устройства Direct3D 9 для работы с 3D-графикой.

Вообще весь этап по инициализации Direct3D 9 можно разделить на три части. Сначала мы создаем объект Direct3D 9, потом совершаем установки параметров представления для Direct3D 9 и в завершение, на основании произведенных установок, создаем устройство Direct3D 9.

Рассмотрим всю функцию `IntialDirect3D()` в целом, поместив в нее код инициализации `Direct3D`, а потом разберем все составляющие данной функции:

```
HRESULT IntialDirect3D(HWND hwnd)
{
    if (NULL == (pDirect3D = Direct3DCreate9(D3D_SDK_VERSION)))
        return E_FAIL;
    D3DDISPLAYMODE Display;
    if (FAILED(pDirect3D ->
        GetAdapterDisplayMode(D3DADAPTER_DEFAULT, Display)))
        return E_FAIL;
    D3DPRESENT_PARAMETERS Direct3DParametr;
    ZeroMemory(&Direct3DParametr, sizeof(Direct3DParametr));
    Direct3DParametr.Windowed = TRUE;
    Direct3DParametr.SwapEffect = D3DSWAPEFFECT_DISCARD;
    Direct3DParametr.BackBufferFormat = Direct3DParametr.Format;
    if (FAILED(pDirect3D -> CreateDevice(D3DADAPTER_DEFAULT,
        D3DDEVTYPE_HAL, hwnd,
        D3DCREATE_HARDWARE_VERTEXPROCESSWG,
        &Direct3DParametr, &pDirect3DDevice)))
        return E_FAIL;
    return S_OK
}
```

В конструкции:

```
if (NULL == (pDirect3D = Direct3DCreate9(D3D_SDK_VERSION)))
```

с помощью функции `Direct3DCreate9()` создается основной указатель на интерфейс `IDirect3D9`. Сама функция всегда использует только один макрос `D3D_SDK_VERSION`, который определен в заголовочном файле `d3d9.h`, указывающий на текущую версию SDK. Функция должна вернуть отличное от `NULL` значение, являющееся указателем на интерфейс `IDirect3D9`. Если это не происходит, то очевидно `Direct3D 9` не установлен на компьютере пользователя.

Далее мы должны получить информацию о текущем режиме визуального отображения дисплея, здесь подразумевается разрешение экрана, цветность и формат поверхности дисплея. Нам необходимо знать эту информацию, для того чтобы использовать, а точнее копировать текущие установки в задний буфер (back buffer). В `Direct3D` все, что выводится на экран, рисуется сначала в заднем буфере, а потом копируется в первичный буфер, т. е. не-

посредственно на экран, этот процесс называется *двойной буферизацией*. Происходит постоянная перерисовка заднего буфера с последующим копированием на экран монитора, что и осуществляет плавную анимацию объектов. В следующей строке:

```
D3DDISPLAYMODE Display;
```

мы объявляем переменную на структуру `D3DDISPLAYMODE`. Она будет использоваться при создании заднего буфера для соответствия текущих установок формата дисплея. Разберем прототип структуры `D3DDISPLAYMODE`:

```
typedef struct _D3DDISPLAYMODE {  
    UINT          Width;  
    UINT          Height;  
    UINT          RefreshRate;  
    D3DFORMAT      Format;  
} DISPLAYMODE;
```

Структура `D3DDISPLAYMODE` имеет следующие параметры:

- ☐ `Width` — ширина рабочей поверхности экрана в пикселах;
- ☐ `Height` — высота рабочей поверхности экрана в пикселах;
- ☐ `RefreshRate` — частота регенерации. Значение 0 указывает на значение по умолчанию;
- ☐ `Format` — формат режима визуального отображения, член перечисляемого типа `D3DFORMAT`. Определяет различные типы поверхностных форматов, представляя собой довольно большую структуру данных, которую мы рассмотрим лишь частично.

Рассмотрим тип `D3DFORMAT`:

```
typedef enum _D3DFORMAT {  
    D3DFMT_UNKNOWN          = 0,  
  
    D3DFMT_R8G8B8           = 20,  
    D3DFMT_A8R8G8B8         = 21,  
    D3DFMT_X8R8G8B8         = 22,  
    D3DFMT_R5G6B5           = 23,  
    D3DFMT_X1R5G5B5         = 24,  
    D3DFMT_A1R5G5B5         = 25,  
    D3DFMT_A4R4G4B4         = 26,  
    D3DFMT_R3G3B2           = 27,  
    D3DFMT_A8               = 28,  
    D3DFMT_A8R3G3B2         = 29,
```

```
D3DFMT_X4R4G4B4      = 30,
D3DFMT_A2B10G10R10   = 31,
D3DFMT_G16R16         = 34,

D3DFMT_A8P8           = 40,
D3DFMT_P8             = 41,

D3DFMT_L8             = 50,
D3DFMT_A8L8           = 51,
D3DFMT_A4L4           = 52,

D3DFMT_V8U8           = 60,
D3DFMT_L6V5U5         = 61,
D3DFMT_X8L8V8U8       = 62,
D3DFMT_Q8W8V8U8       = 63,
D3DFMT_V16U16         = 64,
D3DFMT_W11V11U10      = 65,
D3DFMT_A2W10V10U10    = 67,

D3DFMT_UYVY = MAKEFOURCC('U', 'Y', 'V', 'Y'),
D3DFMT_YUY2 = MAKEFOURCC('Y', 'U', 'Y', '2'),
D3DFMT_DXT1 = MAKEFOURCC('D', 'X', 'T', '1'),
D3DFMT_DXT2 = MAKEFOURCC('D', 'X', 'T', '2'),
D3DFMT_DXT3 = MAKEFOURCC('D', 'X', 'T', '3'),
D3DFMT_DXT4 = MAKEFOURCC('D', 'X', 'T', '4'),
D3DFMT_DXT5 = MAKEFOURCC('D', 'X', 'T', '5'),

D3DFMT_D16_LOCKABLE = 70,
D3DFMT_D32           = 71,
D3DFMT_D15S1         = 73,
D3DFMT_D24S8         = 75,
D3DFMT_D16           = 80,
D3DFMT_D24X8         = 77,
D3DFMT_D24X4S4       = 79,

D3DFMT_VERTEXDATA    = 100,
D3DFMT_INDEX16        = 101,
D3DFMT_INDEX32        = 102,
```

```
D3DFMT_FORCE_DWORD = 0xFFFFFFFF
} D3DFORMAT;
```

Тип `D3DFORMAT` имеет следующие параметры:

- ❑ `D3DFMT_UNKNOWN` — формат поверхности, используемый по умолчанию, мы будем пользоваться как раз этим значением;
- ❑ `D3DFMT_R8G8B8` — 24-битный формат RGB;
- ❑ `D3DFMT_A8R8G8B8` — 32-битный формат ARGB;
- ❑ `D3DFMT_D16_LOCKABLE` — 16-битный формат буфера глубины;
- ❑ `D3DFMT_D32` — 32-битный формат буфера глубины;
- ❑ `D3DFMT_D15S1` — 16-битный формат буфера глубины, где 15 битов резервируются для канала глубины и 1 бит для канала трафарета;
- ❑ `D3DFMT_D24S8` — 32-битный формат буфера глубины, где 24 бита резервируются для канала глубины и 8 битов для канала трафарета;
- ❑ `D3DFMT_D16` — 16-битный формат буфера глубины;
- ❑ `D3DFMT_D24X8` — 32-битный формат буфера глубины, где 24 бита резервируются для канала глубины;
- ❑ `D3DFMT_D24X4S4` — 32-битный формат буфера глубины, где 24 бита резервируются для канала глубины и 4 бита для канала трафарета;
- ❑ `D3DFMT_VERTEXDATA` — формат поверхности буфера вершин;
- ❑ `D3DFMT_INDEX16` — 16-битный индекс буфера глубины;
- ❑ `D3DFMT_INDEX32` — 32-битный индекс буфера глубины;
- ❑ `D3DFMT_FORCE_DWORD` — значение не используется.

К сожалению, `DirectX` содержит большое количество всевозможных вложений, когда параметр функции является каким-нибудь перечисляемым типом, имея много разных значений. С этим ничего не поделаешь, но и пугаться не надо, в процессе обучения и совершенствования своих знаний вы во всем со временем разберетесь. Поверьте мне, я сам сначала хватался за голову, а сейчас не испытываю никаких неудобств по этому поводу.

Далее с помощью функции `IDirect3D9::GetAdapterDisplayMode` мы получаем текущий формат дисплея.

Прототип функции выглядит следующим образом:

```
HRESULT GetAdapterDisplayMode(
    UINT Adapter,
    DISPLAYMODE* pMode);
```

Функция `GetAdapterDisplayMode()` имеет следующие параметры:

- ❑ `Adapter` — видеоадаптер дисплея, значение `D3DADAPTER_DEFAULT` использует первичный адаптер по умолчанию;
- ❑ `pMode` — это указатель на структуру `DISPLAYMODE`, в нашем случае `Display`, который заполняется информацией, описывающей текущий режим адаптера.

Запишем следующее:

```
if (FAILED(pDirect3D
    -> GetAdapterDisplayMode(D3DADAPTER_DEFAULT, &Display)));
return E_FAIL;
```

В этой конструкции используются макрос `FAILED` и код возврата `E_FAIL`. Далее создается объект параметров представления `Direct3DParametr`, заполняя поля которого, можно определить, как будет вести себя наше 3D-приложение:

```
D3DPRESENT_PARAMETERS Direct3DParametr;
```

Следующая строка:

```
ZeroMemory(&Direct3DParametr, sizeof(Direct3DParametr));
```

очищает создаваемую структуру от всевозможного "мусора". Первый параметр — `Direct3DParametr` — это ссылка на параметр представления, иначе говоря: то, что нужно очистить. Второй — `sizeof(Direct3DParametr)` — это размер того, что будет очищаться, т.е. все используемые поля структуры `D3DPRESENT_PARAMETERS`. Сам прототип функции `ZeroMemory()` выглядит следующим образом:

```
VOID ZeroMemory(
    PVOID Destination
    SIZE_T Length);
```

Функция `ZeroMemory()` имеет такие параметры:

- ❑ `Destination` — указатель на начало адреса блока памяти, который необходимо заполнить нулями;
- ❑ `Length` — размер блока памяти в байтах, необходимый для заполнения нулями.

Теперь объектом `Direct3DParameter` структуры `D3DPRESENT_PARAMETERS` можно пользоваться в полной мере путем задания значений полей структуры. Для этого рассмотрим прототип структуры `D3DPRESENT_PARAMETERS` и, в процессе объяснения, сразу заполним поля, необходимые нам для работы:

```
typedef struct _D3DPRESENT_PARAMETERS_ {
    UINT BackBufferWidth;
```

```

UINT                BackBufferHeight;
D3DFORMAT           BackBufferFormat;
UINT                BackBufferCount;
D3DMULTISAMPLE_TYPE MultiSampleType;
D3DSWAPEFFECT       SwapEffect;
HWND                hDeviceWindow;
BOOL                Windowed;
BOOL                EnableAutoDepthStencil;
D3DFORMAT           AutoDepthStencilFormat;
DWORD               Flags;
UINT                FullScreen_RefreshRateInHz;
UINT                FullScreen_PresentationInterval;
} D3DPRESENT_PARAMETERS;

```

В примере мы пока затрагиваем три параметра, в последующих уроках, по мере усложнения материала, будут появляться новые значения, а значит, и больше возможностей:

```

Direct3DParametr.Windowed = TRUE;
Direct3DParametr.SwapEffect = D3DSWAPEFFECT_DISCARD;
Direct3DParametr.BackBufferFormat = Display.Format;

```

Параметры структуры `D3DPRESENT_PARAMETERS` следующие:

- ❑ `Windowed` — это видеорежим, используемый вашим приложением. Значение `TRUE` определяет оконный режим. В данный момент мы работаем именно с ним, на *уроке 1* был указан размер окна — 500×400 пикселей. `FALSE` — это полноэкранный режим, т. е. во весь экран;
- ❑ `BackBufferFormat` — формат поверхности заднего буфера. Задний буфер должен соответствовать текущим настройкам видеорежима, что осуществляется с помощью строки `Display.Format`. Здесь мы обращаемся к переменной `Display`, содержащей текущий формат дисплея. Такая запись не должна вас смущать — это обычная практика в C++ при работе с классами и структурами;
- ❑ `SwapEffect` — этот параметр служит для определения обмена буферов. Перечисляемый тип, член `D3DSWAPEFFECT`. Так, если значение `Windowed` — `TRUE` и параметр `SwapEffect` установлен в `D3DSWAPEFFECT_FLIP`, то создается один задний буфер;
- ❑ `BackBufferWidth` — ширина заднего буфера в пикселах;
- ❑ `BackBufferHeight` — высота заднего буфера в пикселах;
- ❑ `BackBufferCount` — это число задних буферов, их значение может быть 0, 1, 2 или 3. Один задний буфер считается минимальным;

- ❑ `MultiSampleType` — перечисляемый тип, член `D3DMULTISAMPLE_TYPE` и должен быть определен как `D3DMULTISAMPLE_NONE`, если параметр `SwapEffect` не был установлен значением `D3DSWAPEFFECT_DISCARD`;
- ❑ `hDeviceWindow` — если приложение работает в полноэкранном режиме, то задействуется вся поверхность экрана монитора. Если параметр `Windowed` имеет значение `TRUE`, то это оконный режим, если же устанавливается значение `FALSE`, то приложение будет работать в полноэкранном режиме;
- ❑ `EnableAutoDepthStencil` — если этот параметр имеет значение `TRUE`, то для Direct3D 9 становится доступен буфер глубины окна, и устройство Direct3D 9 создаст буфер трафарета;
- ❑ `AutoDepthStencilFormat` — член перечисляемого типа `D3DFORMAT`. Формат автоматической поверхности трафарета созданного устройства. Этот параметр игнорируется, если `EnableAutoDepthStencil` не установлен в значение `TRUE`;
- ❑ `Flags` — этот параметр может быть установлен в 0, или применен флажок `D3DPRESENTFLAG_LOCKABLE_BACKBUFFER`. Этот флажок устанавливается тогда, когда приложение требует блокировать задний буфер;
- ❑ `FullScreen_RefreshRateInHz` — частота обновления экрана;
- ❑ `FullScreen_PresentationInterval` — так обозначается максимальный интервал переключения заднего буфера. Для полноэкранного переключения может использоваться `D3DPRESENT_INTERVAL_DEFAULT` или значение, соответствующее одному из флажков, перечисленных в структуре `D3DCAPS9`. Это очень большая структура и пока нет особого смысла в рассмотрении ее значений.

Структура `D3DPRESENT_PARAMETERS` на первый взгляд кажется довольно сложной, но на самом деле все сводится к подстановке заранее известных значений. Значений не так и много, все они в основном взаимосвязаны, остальное дело практики. Со временем вы обязательно разберетесь во всем более подробно.

Далее мы создаем объект для интерфейса устройства с помощью функции `CreateDevice()`. Здесь я хочу остановиться и быть уверенным, что вы поняли — главный интерфейс `IDirect3D9` создается с помощью функции `Direct3DCreate9()`, а интерфейс устройства Direct3D 9 — функцией `IDirect3DDevice9::CreateDevice`. Это две абсолютно разные функции. Рассмотрим прототип `CreateDevice()`:

```
HRESULT CreateDevice(
    UINT                Adapter,
    D3DDEVTYPE          DeviceType,
    HWND                hFocusWindow,
```

```

DWORD                BehaviorFlags,
D3DPRESENT_PARAMETERS* pPresentationParameters,
IDirect3DDevice9**    ppReturnedDeviceInterface
);

```

Функция `CreateDevice()` имеет следующие параметры:

- ❑ **Adapter** — это ваша видеокарта. Обычно на компьютере установлен один видеоадаптер, поэтому используется значение `D3DADAPTER_DEFAULT`, т. е. по умолчанию первичный видеоадаптер;
- ❑ **DeviceType** — определяет желаемый тип устройства. От этого параметра зависит, будет ли использована акселерация видеокарты. Член перечисляемого типа `D3DDEVTYPE` имеет несколько разных флагов, рассмотрим структуру `D3DDEVTYPE` более подробно: `typedef enum _D3DDEVTYPE {D3DDEVTYPE_HAL = 1, D3DDEVTYPE_REF = 2, D3DDEVTYPE_SW = 3, D3DDEVTYPE_FORCE_DWORD = 0xffffffff} D3DDEVTYPE;`. Структура `D3DDEVTYPE` имеет такие параметры:
 - `D3DDEVTYPE_HAL` — используются возможности аппаратного обеспечения. Выберем данный параметр, чтобы задействовать видеоадаптер для ускорения работы с графикой;
 - `D3DDEVTYPE_REF` — эмулирует программно;
 - `D3DDEVTYPE_SW` — это значение используется вместе с функцией `IDirect3D9::RegisterSoftwareDevice`;
 - `D3DDEVTYPE_FORCE_DWORD` — не используется;
- ❑ **hFocusWindow** — в этом параметре необходимо установить дескриптор главного окна, у нас `hwnd`, чтобы определить, какому из окон принадлежит это значение;
- ❑ **BehaviorFlags** — указывает, как будет происходить обработка вершин, осуществляется при помощи комбинации одного или более флагов:
 - `D3DCREATE_HARDWARE_VERTEXPROCESSING` — в этом случае используется видеокарта и происходит аппаратная обработка вершин. Установим данный флаг, потому что на сегодняшний день вполне приличная видеокарта стоит не дорого и имеет возможность аппаратно реализовать практически любую обработку графики;
 - `D3DCREATE_SOFTWARE_VERTEXPROCESSING` — этот флаг применим в том случае, если вы хотите использовать программную обработку вершин;
- ❑ **pPresentationParameters** — является указателем на структуру `D3DPRESENT_PARAMETERS`, в нашем случае на `Direct3DParameter`. Таким образом, мы используем все присвоенные ранее параметры;

❑ `ppReturnedDeviceInterface` — в этом параметре переменной `pDirect3DDevice` присваивается результат, полученный от функции `CreateDevice()`, т. е. указатель на интерфейс `IDirect3DDevice9`.

Вся операция по созданию объекта интерфейса устройства или просто интерфейса, будет выглядеть следующим образом:

```
pDirect3D -> CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hwnd,
                        D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                        &Direct3DParametr, &pDirect3DDevice)
```

После создания функции `InitialDirect3D()` ее необходимо поместить в `WinMain()` непосредственно после функции `CreateWindowEx()`:

```
if (SUCCEEDED(InitialDirect3D(hwnd)))
{
    ShowWindow(hwnd, SW_SHOWDEFAULT);
    UpdateWindow(hwnd);
    while(GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

Обработка ошибок в DirectX

При работе с DirectX, и не только, необходимо использовать систему обработки ошибок, например, при создании устройства или инициализации Direct3D 9. За всем этим должен следить сам программист. В Direct3D 9 для этих целей определены специальные макросы:

- ❑ `FAILED()` — проверяет код на наличие сбоя;
- ❑ `SUCCEEDED()` — проверяет код на успешность выполнения.

Оба макроса в принципе работают по одной схеме, имея небольшие различия; давайте их рассмотрим. Первый макрос имеет такую структуру:

```
if (FAILED(...))
{сбой}
else
{все хорошо, идем дальше}
```

Второй макрос:

```
if (SUCCEEDED(...))
{все хорошо, идем дальше}
```

```
else
    {сбой}
```

В качестве возвратной точки существуют коды возврата. При какой бы то ни было ошибке, установив определенный код возврата, вы можете получить информацию о природе происхождения ошибки. Кодов возврата очень много, отдельно рассматривать в книге их не имеет смысла, а во всех примерах используется универсальный код возврата `E_FAIL`; подробную информацию о всех кодах возврата можно найти в документации к DirectX 9 SDK.

Рендеринг, или рисуем в окне приложения

Рендеринг — это процесс прорисовки трехмерных объектов. Пока мы еще не научились рисовать 3D-объект, поэтому нарисуем какой-нибудь цвет, а точнее закрасим задний буфер какой-нибудь цветовой компонентой.

Для этих целей создадим функцию `RenderingDirect3D()`, которая будет рисовать в нашем приложении:

```
VOID RenderingDirect3D();
```

Давайте для начала очистим задний буфер функцией `IDirect3DDevice9::Clear`. Рассмотрим ее прототип:

```
HRESULT Clear(
    DWORD          Count,
    CONST D3DRECT* pRects,
    DWORD          Flags,
    D3DCOLOR       Color,
    float          Z,
    DWORD          Stencil
);
```

Функция `Clear()` имеет следующие параметры:

- ❑ `Count` — это номер массива прямоугольников, подвергающихся очистке. Вся область рендеринга состоит из массива прямоугольников. Если указать значение 0, то будет использована вся поверхность, т. е. весь массив прямоугольников или вся область рендеринга;
- ❑ `pRect` — означает адрес массива прямоугольников. Весь массив описывается структурой `D3DRECT`, задающей размеры прямоугольной области. Рассмотрим запись данной структуры: `typedef struct _D3DRECT {LONG x1; LONG y1; LONG x2; LONG y2;} D3DRECT;`

Структура D3DRECT имеет такие параметры:

- x_1 и y_1 — координаты верхнего левого угла прямоугольника;
- x_2 и y_2 — координаты правого нижнего угла прямоугольника;
- **Flags** — параметр, определяющий флаги, указывающие, какие из поверхностей должны быть очищены. Этим параметром может быть любая комбинация из следующих флагов:
 - D3DCLEAR_STENCIL — очистит буфер трафарета до значения в параметре Stencil;
 - D3DCLEAR_TARGET — очистит буфер поверхности до цвета в параметре Color. Воспользуемся данным флагом для очистки буфера поверхности в заданный цвет;
 - D3DCLEAR_ZBUFFER — очистит буфер глубины до значения в параметре z;
- **Color** — в этом параметре используется обычный макрос цвета D3DCOLOR_XRGB(255, 255, 50), в данном случае желтого, который и закрасит буфер цветом;
- **z** — этот параметр задает значения для Z-буфера. Значение может находиться в диапазоне от 0.0 — это самое близкое расстояние для средства просмотра и 1.0 — дальнее расстояние. Применим значение 1.0f и оставим пока этот параметр без детального рассмотрения, потому как один из уроков будет посвящен непосредственно буферу глубины;
- **Stencil** — значение буфера трафарета, может быть в диапазоне от 0 до $2^n - 1$, где n разрядная глубина буфера трафарета. Этот параметр тоже будет объяснен чуть позже, а пока воспользуемся значением 0.

Вообще, поскольку мы пользуемся флагом D3DCLEAR_TARGET, то значения в параметрах z и Stencil будут игнорироваться. Если по необходимости подставлять другие флаги либо их комбинацию, то значения, не используемые в параметрах Color, z, Stencil, также будут игнорироваться.

Итак, зная все необходимые значения параметров функции Clear(), можно очистить задний буфер:

```
pDirect3DDevice -> Clear(0, NULL, D3DCLEAR_TARGET,  
                        D3DCOLOR_XRGB(255, 255, 50), 1.0f, 0);
```

В Direct3D существует такое понятие, как *сцена* (scene), в нее входят все 3D-объекты со своими различными свойствами. Перед выводом сцены вызывается функция BeginScene() из интерфейса IDirect3DDevice9, она как бы сигнализирует о том, что можно начинать рендеринг объектов сцены, т. е. вы даете фактически команду BeginScene() для прорисовки сцены:

```
pDirect3DDevice -> BeginScene();    // начало сцены
```

Сама сцена заканчивается функцией `EndScene()` из интерфейса `IDirect3DDevice9`:

```
pDirect3DDevice -> EndScene(); // конец сцены
```

Обе функции не имеют параметров. Поскольку у нас нет пока сцены (она в данный момент пуста), то между этими двумя функциями ничего нет, но за каждым вызовом функции `BeginScene()` обязательно должна следовать `EndScene()`, а между этими двумя функциями происходит рендеринг. Теперь, все что мы прорисовали в заднем буфере, выведем на экран с помощью функции `IDirect3DDevice9::Present`, которая копирует все нарисованное из заднего буфера в передний, т. е. на экран. Посмотрим внимательно на прототип функции:

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

Функция `Present()` имеет следующие параметры:

- ❑ `pSourceRect` — первый параметр — это указатель на структуру `RECT` исходной поверхности. Если использовать значение `NULL` (пустой указатель), то будет представлена полная исходная поверхность;
- ❑ `pDestRect` — указатель на структуру `RECT` поверхности адресата. Чтобы использовать всю поверхность, нужно поставить значение `NULL`, т. о. копируется содержимое заднего буфера "один в один" в передний;
- ❑ `hDestWindowOverride` — указатель на окно адресата, чья клиентская область взята как адрес для этого параметра представления. Выставив значение в `NULL`, мы тем самым говорим, что используем поле `hWndDeviceWindow` в структуре `D3DPRESENT_PARAMETERS`. Напомню, в `hWndDeviceWindow` дескриптором окна был задан параметр `hwnd` — метка созданного нами окна;
- ❑ `pDirtyRegion` — последний параметр, "отголосок" прежних версий `DirectX` и уже не используется, а оставлен просто для совместимости, поэтому выставляется всегда значением `NULL`.

Весь вызов функции `Present()` сводится к пустому протокольному мероприятию. Что же, тем лучше для нас:

```
pDirect3DDevice -> Present(NULL, NULL, NULL, NULL);
```

А теперь сложим все рассмотренное выше вместе, получив тем самым функцию рендеринга `RenderingDirect3D()`:

```
VOID RenderingDirect3D()
{
    pDirect3DDevice -> Clear(0, NULL, D3DCLEAR_TARGET,
                           D3DCOLOR_XRGB(255, 255, 0), 1.0f, 0);
    pDirect3DDevice -> BeginScene();
    // здесь происходит прорисовка сцены
    pDirect3DDevice -> EndScene();
    pDirect3DDevice -> Present(NULL, NULL, NULL, NULL);
}
```

Создав функцию, которая будет осуществлять рендеринг, поместим ее в очередь функции `MainWinProc()` в то место, где происходит прорисовка окна:

```
case WM_PAINT:
    RenderingDirect3D();
    ValidateRect(hwnd, NULL);
    return 0;
```

Функция `ValidateRect(hwnd, NULL)` вызывается для прорисовки всей клиентской области окна. Первый параметр — `hwnd` — дескриптор нашего окна, а второй — `NULL` — служит для перерисовки по умолчанию, т. е. всего окна. На этом весь процесс рендеринга закончен. Скажу только, что нами был рассмотрен самый простейший способ с использованием функции `MainWinProc()` и сообщения `WM_PAINT`. На следующих уроках мы рассмотрим другие способы вызова функции `RenderingDirect3D()`.

Освобождаем ресурсы, захваченные Direct3D

В тот момент, когда ваше приложение будет закрыто, оно автоматически должно вернуть все захваченные Direct3D 9 ресурсы системы, и вы должны об этом позаботиться. По правилам использования COM-интерфейсов ссылки должны обнуляться и выгружаться (удаляться) из памяти. Это просто, в вашем распоряжении имеется функция `Release()`, которую мы рассмотрели на уроке 2, она и позаботится обо всем. Все, что нужно помнить, это с какой очередностью должно происходить освобождение объектов, а именно — в порядке убывания. Последним мы объявили интерфейс устройства `pDirect3DDevice9`, его удаляем первым. И так по возрастающей в обратном порядке.

Создадим для этих целей функцию, которую назовем `DeleteDirect3D()`, осуществляющую освобождение захваченных ресурсов:

```
VOID DeleteDirect3D()
{
    if (pDirect3DDevice! = NULL)
        pDirect3DDevice -> Release();
    if (pDirect3D! = NULL)
        pDirect3D -> Release();
}
```

Поместим эту функцию в `MainWinProc()` после сообщения `WM_DESTROY`, где `DeleteDirect3D()` становится в очередь на удаление:

```
case WM_DESTROY:
    DeleteDirect3DD();
    PostQuitMessage(0);
    return 0;
```

Посмотрим, что у нас в итоге получилось, в листинге 3.1.

Листинг 3.1. Файл `InitialDirect3D.cpp`

```
//-----
//  InitialDirect3D.cpp
//  инициализируем Direct3D
//-----
#include <windows.h>    // подключаем заголовочный файл Windows
#include <d3d9.h>        // подключаем заголовочный файл DirectX 9 SDK
//-----
//  глобальные переменные
//-----
LPDIRECT3D9 pDirect3D          = NULL;
LPDIRECT3DDEVICE9 pDirect3DDevice = NULL;
//-----
//  функция
//  InitialDirect3D()
//  инициализация Direct3D
//-----
HRESULT InitialDirect3D(HWND hwnd)
{
```

```

    if (NULL == (pDirect3D = Direct3DCreate9(D3D_SDK_VERSION)))
        return E_FAIL;
    D3DDISPLAYMODE Display;
    if (FAILED(pDirect3D ->
        GetAdapterDisplayMode(D3DADAPTER_DEFAULT, &Display)))
        return E_FAIL;
    D3DPRESENT_PARAMETERS Direct3DParametr;
    ZeroMemory(&Direct3DParametr, sizeof(Direct3DParametr));
    Direct3DParametr.Windowed = TRUE;
    Direct3DParametr.SwapEffect = D3DSWAPEFFECT_DISCARD;
    Direct3DParametr.BackBufferFormat = Display.Format;

    if (FAILED(pDirect3D ->
        CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
            hwnd, D3DCREATE_SOFTWARE_VERTEXPROCESSING,
            &Direct3DParametr, &pDirect3DDevice)))
        return E_FAIL;
    return S_OK;
}

//-----
// функция
// RenderingDirect3D()
// рисование
//-----
VOID RenderingDirect3D()
{
    if (pDirect3DDevice == NULL) // проверяем ошибки
        return;
    pDirect3DDevice -> Clear(0, NULL, D3DCLEAR_TARGET,
        D3DCOLOR_XRGB(255, 255, 0), 1.0f, 0);
    pDirect3DDevice -> BeginScene(); // начало сцены
    // здесь происходит прорисовка сцены
    pDirect3DDevice -> EndScene(); // конец сцены
    pDirect3DDevice -> Present(NULL, NULL, NULL, NULL);
}

//-----
// функция
// DeleteDirect3D()
// освобождение захваченных ресурсов

```

```

//-----
VOID DeleteDirect3D()
{
    if (pDirect3DDevice != NULL)
        pDirect3DDevice -> Release();
    if (pDirect3D != NULL)
        pDirect3D -> Release();
}
//-----
// функция
// MainWinProc()
// обработка сообщений
//-----

LRESULT CALLBACK MainWinProc(HWND    hwnd,
                                UINT   msg,
                                WPARAM  wparam,
                                LPARAM  lparam)
{
    switch(msg)
    {
        case WM_PAINT:
            RenderingDirect3D();
            ValidateRect(hwnd, NULL);
            break;
        case WM_DESTROY:
            {
                DeleteDirect3D();
                PostQuitMessage(0);
                return(0);
            } break;
    }
    return (DefWindowProc(hwnd, msg, wparam, lparam));
}
//-----
// функция
// WinMain
// входная точка приложения

```

```
//-----
int WINAPI WinMain( HINSTANCE  hinstance,
                   HINSTANCE  hprevinstance,
                   LPSTR      lp cmdline,
                   int         ncmdshow)
{
    WNDCLASSEX  windowclass;    // создаем класс
    HWND        hwnd;           // создаем дескриптор окна
    MSG         msg;            // идентификатор сообщения

    // определим класс окна WNDCLASSEX
    windowclass.cbSize = sizeof(WNDCLASSEX);
    windowclass.style = CS_DBLCLKS|CS_OWNDC|CS_HREDRAW|CS_VREDRAW;
    windowclass.lpfnWndProc = MainWinProc;
    windowclass.cbClsExtra = 0;
    windowclass.cbWndExtra = 0;
    windowclass.hInstance = hinstance;
    windowclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    windowclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    windowclass.hbrBackground= (HBRUSH)GetStockObject(GRAY_BRUSH);
    windowclass.lpszMenuName = NULL;
    windowclass.lpszClassName = "WINDOWSCCLASS";
    windowclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

    // регистрируем класс
    if (!RegisterClassEx(&windowclass))
        return(0);

    // теперь, когда класс зарегистрирован, можно создать окно
    if (!(hwnd = CreateWindowEx(NULL, "WINDOWSCCLASS",
        "Инициализация Direct3D",
        WS_OVERLAPPEDWINDOW|WS_VISIBLE, 300, 150,
        500, 400, NULL, NULL, hinstance, NULL)))
        return(0);

    if (SUCCEEDED(InitialDirect3D(hwnd)))
    {
        ShowWindow(hwnd, SW_SHOWDEFAULT);
        UpdateWindow(hwnd);
        while( GetMessage(&msg, NULL, 0, 0) )
        {

```

```
        TranslateMessage (&msg);  
        DispatchMessage (&msg);  
    }  
}  
return 0;  
}
```

Итоги урока

Весь урок был посвящен инициализации и настройке Direct3D путем вызова множества функций и различных интерфейсов, определенных в Direct3D 9. Вы получили ответ на то, как происходит прорисовка объекта на экране монитора, но пока самого объект еще нет, поэтому задний буфер Direct3D был окрашен в желтый цвет. На *уроке 4* мы создадим простой треугольник и прорисуем его на экране.

Урок 4



Рисуем 2D-объект

На *уроке 3*, в созданном окне, мы проинициализировали Direct3D 9 и закрасили, а точнее закрасили в желтый цвет задний буфер. На этом уроке мы научимся рисовать простой двумерный объект. Сначала это будет треугольник, затем квадрат. Весь процесс создания и прорисовки двумерной геометрической фигуры происходит в три этапа:

- установка формата вершин;
- создание буфера вершин;
- рендеринг объекта.

Любой объект, рисуемый на экране, имеет свои координаты в пространстве. Если мы возьмем треугольник, то у него есть три угла, задав координаты которых, можно вывести треугольник на экран. Координаты одного угла задаются с помощью трех величин: X , Y , Z , а данная совокупность значений называется *вершиной*. Сейчас мы не будем использовать матричные преобразования, эта тема будет рассмотрена на двух последующих уроках, а на этом уроке будет использован формат преобразованных вершин. Система координат, применяемая для рендеринга преобразованных вершин, немного отличается от системы координат, используемой при матричных преобразованиях. Начало координат — точка 0.0 , находится в левой верхней точке дисплея. Ось X идет от этой точки горизонтально слева направо по верхней кромке экрана, ось Y опускается вниз, а Z не используется в полной мере, но присутствует при описании вершин. На рис. 4.1 показана система координат, использующаяся в формате преобразованных вершин.

Как мы уже выяснили, любой объект имеет свои координаты в пространстве, а сам объект в Direct3D задается определенным количеством вершин, между которыми и строятся треугольники. Задав координаты вершин, мы можем построить любой объект, например, квадрат, состоящий из двух треугольников, или прямоугольник, содержащий в себе два и более квадратов, и т. д.

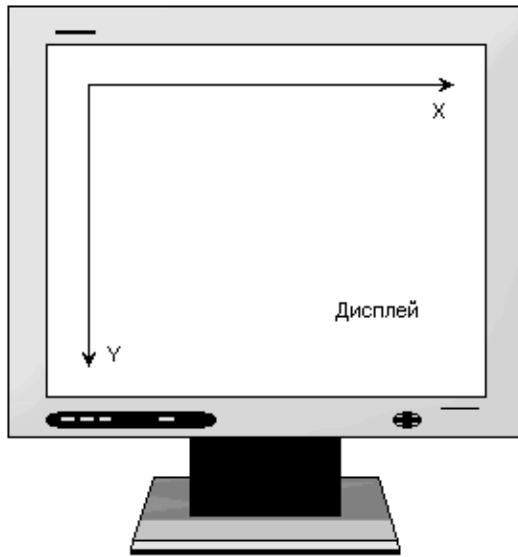


Рис. 4.1. Система координат

Установка формата вершин

Для того чтобы использовать вершины в прорисовке, сначала надо определить тип вершин. С этой целью создается структура, определяющая формат вершин. Все дело в том, что вся трехмерная графика в Direct3D в основном строится на матричных преобразованиях, с помощью которых имеется возможность перемещать, вращать, удалять, приближать и производить другие операции с объектом. И что самое главное, правильно представлять трехмерный объект на двумерной плоскости монитора. Любой объект, как мы знаем, строится из вершин. В матричных преобразованиях весь объект, а точнее каждая из вершин умножается на ту или иную матрицу и представляется на экране монитора. Все вершины объекта имеют непреобразованный формат записи, потому что преобразовываются с помощью матриц. Сейчас мы не используем матрицы, поэтому должны сами задать формат для определения вершин, чтобы Direct3D знал, как представить вершины. Поэтому используется так называемый формат преобразованных вершин, задающийся, например, таким образом:

```
struct CUSTOMVERTEX
{
    FLOAT   X, Y, Z;    // координаты в трехмерном пространстве
    FLOAT   rhw;        // параметр преобразованной вершины
```

```
DWORD   color;        // цвет
FLOAT   tu, tv         // координаты текстур
}
```

С помощью такой структуры можно задать координаты для вершин — X , Y , Z ; rhw — это параметр преобразованной вершины; цвет вершины — `color` и координаты текстур — `tu`, `tv`. Параметры `tu` и `tv` сейчас не используются и будут рассмотрены на *уроке 10*, посвященном текстурам.

Если вы еще не создали новый проект, то самое время это сделать. Откройте Visual C++ .NET, создайте пустой проект `Urok4` и добавьте в него пустой файл, назвав его `Треугольник.cpp`. Скопируйте с предыдущего *урока 3* весь код, приведенный на компакт-диске в папке `Code\Urok3`, и вставьте его в только что созданный файл.

Примечание

Не забывайте добавлять в новый созданный проект библиотеку `d3d9.lib`, как это было сделано на *уроке 3*. Иначе возникнут ошибки при компиляции проекта.

В глобальных переменных объявим структуру `CUSTOMVERTEX`, задающую формат вершин:

```
struct CUSTOMVERTEX
{
    FLOAT   X, Y, Z, rhw;
    DWORD   color;
}

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW|D3DFVF_DIFFUSE)
```

Структура `CUSTOMVERTEX` имеет следующие параметры:

- ❑ `X`, `Y`, `Z` — координаты по осям;
- ❑ `rhw` — этим параметром задается трансформированная или преобразованная вершина для двумерных координат;
- ❑ `color` — цвет, в который закрашиваются вершины. Пока мы пользуемся, скажем так, локальным освещением, просто освещая вершину цветом и не используя освещение `Direct3D`.

С помощью строки:

```
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW|D3DFVF_DIFFUSE)
```

описывается формат содержания вершин в отдельном потоке данных. Здесь применяется гибкий формат вершин — FVF (flexible vertex format), задача которого состоит в определении содержания формата вершин. Заметьте, как все перечисленные элементы совпадают с объявлениями структуры `CUSTOMVERTEX`. Дело в том, что эта комбинация флагов и описывает, как от-

форматирована ваша вершина. Можно применять различные комбинации флагов для определения формата вершин. Вся спецификация FVF имеет небольшое количество флагов:

- ❑ `D3DFVF_DIFFUSE` — для параметра `color` показывает, что применяется цветовая составляющая с рассеянным светом;
- ❑ `D3DFVF_NORMAL` — формат вершин, включает вектор нормали вершин. Не может применяться с флагом `D3DFVF_XYZRHW`. Этот флаг используется при работе с материалом и светом;
- ❑ `D3DFVF_XYZ` — формат вершин, включает позицию непреобразованной вершины. Не может применяться вместе с флагом `D3DFVF_XYZRHW`. Этим флагом мы и будем пользоваться в трехмерной системе координат;
- ❑ `D3DFVF_XYZRHW` — преобразованный формат вершин. Не может применяться с флагом `D3DFVF_XYZ`;
- ❑ `D3DFVF_XYZW` — формат вершин, содержит преобразованные вершины.

Итак, `D3DFVF_XYZRHW` — это формат преобразованной вершины, а `D3DFVF_DIFFUSE` — цвет для окрашивания вершин. После того как мы разобрались с форматом вершин, можно переходить к следующему этапу по созданию буфера вершин.

Создание буфера вершин

Буфер вершин в определении `Direct3D` — это объект, используемый для хранения вершин. Это просто отличная идея и ничего сложного нет. В общем виде все выглядит так: вы создаете буфер вершин, в котором храните вершины, а также всю сопутствующую информацию о вершинах (цвет, нормали, текстуры, координаты) и в итоге у вас все лежит в одном месте и всегда под рукой. Механизм создания вершин тоже очень прост. Вначале создается переменная, в которой будет храниться указатель на буфер вершин, а уже в созданную переменную записывается этот самый указатель, после чего буфер вершин заполняется данными. Заполнение буфера вершин данными происходит следующим образом: сначала буфер блокируется, потом в буфер копируются данные и в конце буфер разблокируется. После чего в вашем распоряжении имеется полноценный работоспособный буфер вершин, содержащий в себе данные о треугольнике. Давайте рассмотрим все это более подробно и создадим свой буфер вершин.

В `Direct3D` для этих целей специально определен интерфейс `IDirect3DVertexBufferr9`. В файле `Treugolnik.cpp` в глобальных переменных запишем:

```
LPDIRECT3DVERTEXBUFFER9 pBufferVershin = NULL;
```

Этой строкой мы создали указатель `pBufferVershin`, не забыв выставить его значение в ноль. Все дальнейшие действия сводятся к созданию функции, отвечающей за инициализацию буфера вершин, в который мы поместим координаты трех вершин треугольника. Далее создадим сам буфер вершин и заполним его данными о наших вершинах и в конце, уже в функции `RenderingDirect3D()`, в месте, где происходит прорисовка сцены, будем рисовать с помощью созданного ранее буфера вершин, поместив функцию `RenderingDirect3D()` в `WinMain()` для обработки.

Создадим функцию `InitialBufferVershin()`:

```
HRESULT InitialBufferVershin()
```

В самом начале опишем структуру, в которой будем хранить координаты и цвет вершин треугольника:

```
CUSTOMVERTEX Vershin[] =
{
    {300.0f, 300.0f, 0.5f, 1.0f, 0x00000fff}, // A
    {150.0f, 300.0f, 0.5f, 1.0f, 0x00000fff}, // B
    {150.0f, 150.0f, 0.5f, 1.0f, 0x00000fff}, // C
    // X      Y      Z      rhw   цвет
};
```

Первые два значения — это соответственно координаты по осям *X* и *Y*. Не забывайте, что все они определены как числа с плавающей точкой или вещественные числа, поэтому используется, например, запись вида `300.0f`. Все значения исчисляются в пикселах, кроме цвета, конечно. Координата по оси *Z* задается значением `0.5f`; по сути, она сейчас не имеет для нас никакого значения, далее указывается `rhw`, равное `1.0f`. И последнее, это синий цвет, заданный в восьмеричном счислении.

Из всего этого следует, что первая вершина треугольника, а у нас это точка *A*, имеет следующие значения:

```
X = 300.0f
Y = 300.0f
Z = 0.5f
rhw = 1.0f
color = 0x00000fff
```

Прежде чем идти дальше, давайте сразу разберемся, как будет выглядеть рисуемый треугольник на экране после компиляции. Почему-то сразу вспомнился старый программистский анекдот: "Один программист спрашивает у другого:

— Что пишем?

А тот отвечает:

— Откомпилим и узнаем".

Зная все координаты, можем предположить, как будет выглядеть треугольник на экране, схематично представленный на рис. 4.2.

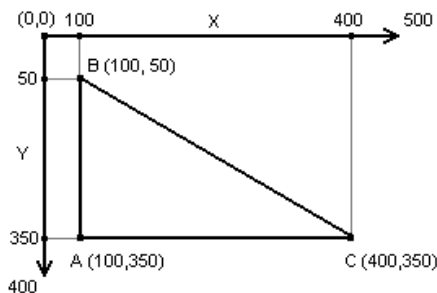


Рис. 4.2. Треугольник с заданными координатами вершин

После того как структура `Vertex` была заполнена всеми данными, необходимо создать буфер вершин. Функция

`IDirect3DDevice9::CreateVertexBuffer`

предназначена для создания данного буфера. Прототип этой функции выглядит так:

```
HRESULT CreateVertexBuffer(
    UNIT          Length,
    DWORD         Usage,
    DWORD         FVF,
    D3DPool       Pool,
    IDirect3DVertexBuffer9**ppVertexBuffer,
    HANDLE*       pHandle);
```

Функция `CreateVertexBuffer()` имеет следующие параметры:

- ❑ `Length` — это размер буфера вершин в байтах, он должен быть не меньше той самой структуры `Vertex`, которая будет храниться в нем. Запись `3*sizeof(CUSTOMVERTEX)` будет справедлива при условии, что 3 — это количество вершин (у нас это *A*, *B*, *C*). Если вы захотите хранить в буфере вершин больше данных, например, в структуре `Vertex` вы инициализировали два треугольника — 6 вершин, или три треугольника — 9 вершин, то соответственно число, стоящее перед оператором `sizeof`, должно быть увеличено на 6 или 9, в зависимости от количества вершин рисуемого примитива;
- ❑ `Usage` — это способ применения буфера вершин. В большинстве случаев устанавливается нулевое значение, но можно воспользоваться одной

или несколькими константами `D3DUSAGE`. Правда, значение константы придется согласовывать с параметром `BehaviorFlags` из `IDirect3D9::CreateDevice`, но пока мы будем пользоваться значением по умолчанию 0. А когда вы уже хорошо освоитесь в `DirectX`, то сможете совершенствовать свой код сколько угодно, благо для этого очень много возможностей;

- `FVF` — здесь указывается формат вершин, установленный ранее, с помощью спецификатора `D3DFVF`. Для наших вершин это `D3DFVF_CUSTOMVERTEX`;
- `Pool` — в этом параметре используется флаг `D3DPOOL_DEFAULT`, описывающий соответствующий формат буфера вершин для размещения его в памяти. Сам параметр `Pool` является членом перечисляемого типа `D3DPOOL`, определяющего класс памяти для содержания буфера ресурса (под буфером ресурса подразумевается буфер вершин индексный буфер и т. д.). Тип `D3DPOOL` имеет следующий вид:

```
tupedef enum D3DPOOL {D3DPOOL_DEFAULT = 0,
D3DPOOL_MANAGED = 1, D3DPOOL_SYSTEMMEM = 2, D3DPOOL_SCRATCH = 3, D3DPOOL_FORCE_DWORD = 0xffffffff} D3DPOOL;
```

Тип `D3DPOOL` включает такие параметры:

- `D3DPOOL_DEFAULT` — значение по умолчанию, применяется чаще всего;
 - `D3DPOOL_MANAGED` — ресурсы копируются автоматически в доступную для устройства память. Полезная константа, с ее помощью можно избежать потерю устройства, потому что управление происходит системной памятью и не обновляется при потере устройства, а значит, всегда будет видеть это устройство;
 - `D3DPOOL_SYSTEMMEM` — использует системную оперативную память;
 - `D3DPOOL_SCRATCH` — ресурс помещается в системную память и не обновляется при потере устройства;
 - `D3DPOOL_FORCE_DWORD` — значение не используется;
- `ppVertexBuffer` — это указатель, в котором будет храниться адрес создаваемого буфера вершин, у нас это `pBufferVerшин`;
 - `pHandle` — всегда устанавливайте значение 0, это зарезервированный параметр.

Теперь, имея всю необходимую информацию, можно создать сам буфер вершин:

```
if (FAILED(pDirect3DDevice -> CreateVertexBuffer(3*sizeof(CUSTOMVERTEX),
0, D3DFVF_CUSTOMVERTEX, D3DPOOL_DEFAULT, &pBufferVerшин, NULL))
return E_FAIL;
```

Создание буфера вершин обязательно сопроводите проверкой на ошибки с помощью макроса `FAILED`. Далее, после того как буфер вершин создан, не-

обходимо скопировать все данные о вершинах в буфер вершин. С помощью функции `IDirect3DVertexBuffer::Lock` происходит блокировка диапазона данных вершин для получения указателя на память буфера вершин. Заблокировав буфер вершин, можно копировать данные с помощью функции `memcpy()`. После того как все данные скопированы или помещены в буфер вершин, необходимо вызвать функцию `IDirect3DVertexBuffer::Unlock` для разблокировки буфера вершин. На каждый вызов функции блокировки буфера `Lock()` вы обязаны вызывать функцию разблокировки `Unlock()`.

Теперь давайте подробно разберемся со всеми перечисленными функциями — первая у нас идет `Lock()`. Прототип данной функции имеет вид:

```
HRESULT Lock(  
    UNIT    OffsetToLock,  
    UNIT    SizeToLock,  
    VOID**  ppbData,  
    DWORD   Flags)
```

Параметры функции `Lock()` следующие:

- ❑ `OffsetToLock` — указывает на границу данных вершин для блокировки. Измеряется в байтах. Для того чтобы заблокировать весь буфер, устанавливается значение 0. Это своего рода смещение, т. е. то, от какого значения нужно блокировать буфер;
- ❑ `SizeToLock` — этим параметром определяется размер блокировки буфера вершин: `sizeof(Vershin)`. В первом параметре мы говорим, откуда блокировать, а во втором, на сколько блокировать. Кстати, оба параметра можно выставить в `NULL`, этим вы заблокируете весь буфер вершин;
- ❑ `ppbData` — означает адрес указателя на указатель с данными буфера вершин, необходим при использовании функции копирования `memcpy()`. Временную переменную нужно создать перед тем, как задействовать функцию `Lock()`;
- ❑ `Flags` — описывает тип осуществляемой блокировки. В большинстве случаев это комбинация из нулей или пяти различных флагов для описания типа блокировки. Могут использоваться флаги:
 - `D3DLOCK_DISCARD` — блокируется каждое положение в пределах блокируемой области. Это распространенный флаг и он используется также при работе с буфером вершин, индексным буфером и текстурами;
 - `D3DLOCK_NO_DIRTY_UPDATE` — при использовании этого флага происходит своего рода добавление данных в память;
 - `D3DLOCK_NO_SYSLock` — этот флаг блокирует систему на долгий промежуток времени, в течение которого можно исполнять и другие режимы работы;

- D3DLOCK_READONLY — приложение не будет записано в буфер;
- D3DLOCK_NOOVERWRITE — этот флаг исключает запись в буфер вершин при уже имеющихся данных в буфере.

Запишем следующее:

```
VOID* pBV;
if (FAILED(pBufferVershin -> Lock(0, sizeof(Vershin), (void**)&pBV, 0)))
return E_FAIL;
```

Здесь pBV — указатель на указатель с данными вершин, временная переменная.

После блокировки буфера вершин можно приступить к копированию данных:

```
memcpy(pBV, Vershin, sizeof(Vershin));
```

И последний, завершающий, шаг — это разблокировка буфера вершин. Она осуществляется функцией `Unlock()`, не имеющей никаких параметров. Просто запомните — заблокировав буфер, его необходимо разблокировать.

```
PBufferVershin -> Unlock();
```

Вся функция целиком имеет такую запись:

```
HRESULT InitialBufferVershin()
{
    CUSTOMVERTEX Vershin[] =
    {
        {300.0f, 300.0f, 0.5f, 1.0f, 0x00000fff}, // A
        {150.0f, 300.0f, 0.5f, 1.0f, 0x00000fff}, // B
        {150.0f, 150.0f, 0.5f, 1.0f, 0x00000fff}, // C
        // X      Y      Z   rhw  цвет
    };
    if (FAILED(pDirect3DDevice -> CreateVertexBuffer(
                                                3*sizeof(CUSTOMVERTEX),
                                                0, D3DFVF_CUSTOMVERTEX,
                                                D3DPOOL_DEFAULT,
                                                &pBufferVershin, NULL)))
        return E_FAIL;
    VOID* pBV;
    if (FAILED(pBufferVershin ->
        Lock(0, sizeof(Vershin), (void**)&pBV, 0)))
        return E_FAIL;
    memcpy(pBV, Vershin, sizeof(Vershin)); // копирование
    pBufferVershin -> Unlock();           // разблокирование
    return S_OK;
}
```

Рендеринг объекта

Последним завершающим этапом, будет вывод прорисованного треугольника на экран. В функции `RenderingDirect3D()`, в том месте, где должна происходить прорисовка сцены, между `BeginScene()` и `EndScene()`, нарисуем наш треугольник. Все действия сводится к последовательному вызову трех функций:

- ❑ `IDirect3DDevice9::SetStreamSource`
- ❑ `IDirect3DDevice9::SetFVF`
- ❑ `IDirect3DDevice9::DrawPrimitive`

После чего, итоговую функцию `RenderingDirect3D()`, отдадим "на растерзание" `WinMain()`.

Первой вызывается функция `SetStreamSource()`. С ее помощью буфер вершин связывается с потоком данных устройства, создавая тем самым, некую ассоциацию между данными вершины и одним потоком данных. Рассмотрим ее прототип:

```
HRESULT SetStreamSource(
    UINT                StreamNumber,
    IDirect3DVertexBuffer* pStreamData,
    UINT                OffsetInBytes,
    UINT                Stride)
```

Функция `SetStreamSource()` имеет следующие параметры:

- ❑ `StreamNumber` — определяет поток данных в диапазоне от 0 до -1. У нас один поток, поэтому установим значение 0;
- ❑ `pStreamData` — это указатель на создаваемый буфер вершин, который связывается с потоком данных `pBufferVerшин`;
- ❑ `OffsetInBytes` — смещение от начала потока, до начала данных вершин. Измеряется в байтах. Здесь вы указываете, с какой вершины начинается вывод. Если поставить 0, то вывод будет происходить от первой вершины;
- ❑ `Stride` — это своего рода "шаг в байтах", от одной вершины до другой. Здесь вы указываете размер структуры, в нашем случае `sizeof(CUSTOMVERTEX)`.

В итоге получаем работоспособный вызов функции `SetStreamSource()`:

```
pDirect3DDevice -> SetStreamSource(0, pBufferVerшин, 0,
                                   sizeof(CUSTOMVERTEX));
```

Следующей идет функция `SetFVF()`, с помощью которой устанавливается формат вершин. Прототип этой функции имеет вид:

```
HRESULT SetFVF(DWORD FVF);
```

Функция имеет только один параметр FVF, содержащий формат преобразованной вершины, установленный ранее:

```
pDirect3DDevice -> SetFVF(D3DFVF_CUSTOMVERTEX);
```

И после того, как вы установили поток источника данных и определили формат типа вершин, можно произвести вывод самого объекта функцией `DrawPrimitive()`. С ее помощью происходит вывод примитива на экран.

Примитив — это любая геометрическая фигура, например, линия, точка, треугольник и т. д. Так вот, эта функция выводит последовательность неиндексированных геометрических примитивов, т. е. наш треугольник. Почему неиндексированных — дело в том, что у нас сейчас так называемый двумерный случай, мы выводим плоскую фигуру на экран, имея всего три вершины, но когда на *уроке 6* будет рисоваться куб, вершин станет в шесть раз больше и для того, чтобы окончательно в них не запутаться и что главное, сэкономить память, применяется концепция индексации вершин. Но это потом, а пока рассмотрим прототип функции:

```
HRESULT DrawPrimitive(  
    D3DPRIMITIVETYPE PrimitiveType,  
    UNIT StartVertex,  
    UNIT PrimitiveCount)
```

Прототип функции `DrawPrimitive()` имеет следующие параметры:

□ `PrimitiveType` — это перечисляемый тип члена `D3DPRIMITIVETYPE`, описывающего тип выводимого на экран примитива. Рассмотрим флаги, используемые в структуре `D3DPRIMITIVETYPE`:

- `D3DPT_POINTLIST` — определяет вершину, как коллекцию изолированных точек;
- `D3DPT_LINELIST` — определяет вершину, как список прямых линий;
- `D3DPT_LINESTRIP` — определяет вершину, как отдельную ломаную линию;
- `D3DPT_TRIANGLELIST` — определяет указанную вершину, как последовательность изолированных треугольников. Каждая группа из трех вершин определяет отдельный треугольник. Используем этот флаг для определения списка треугольников;
- `D3DPT_FORCE_DWORD` — это значение не используется.

□ `StartVertex` — индекс первой вершины для загрузки (0);

□ `PrimitiveCount` — своего рода счетчик, указывающий на количество выводимых треугольников на экран. Если рисуем один треугольник, значит, ставим цифру 1, два треугольника — 2 и т. д.

Зная все параметры функции `DrawPrimitive()`, напомним:

```
pDirect3DDevice -> DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);
```

Функция рендеринга будет иметь следующий вид:

```
VOID RenderingDirect3D()
{
    if (pDirect3DDevice == NULL)          // проверяем ошибки
        return;
    pDirect3DDevice -> Clear(0, NULL, D3DCLEAR_TARGET,
                           D3DCOLOR_XRGB(0, 255, 0), 1.0f, 0);
    pDirect3DDevice -> BeginScene();      // начало сцены
    // здесь происходит прорисовка сцены
    pDirect3DDevice -> SetStreamSource(0, pBufferVershin, 0,
                                       sizeof(CUSTOMVERTEX));
    pDirect3DDevice -> SetFVF(D3DFVF_CUSTOMVERTEX);
    pDirect3DDevice -> DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);
    pDirect3DDevice -> EndScene();        // конец сцены
    pDirect3DDevice -> Present(NULL, NULL, NULL, NULL);
}
```

Для вывода всего нарисованного на экран необходимо поместить функцию `RenderingDirect3D()` в главный цикл событий `WinMain()`:

```
while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
RenderingDirect3D();
```

И если поступить именно так, то все, что вы увидите после успешной компиляции, — это мигнувшее на доли секунды окно. Дело в том, что сама функция `GetMessage()` не предназначена для работы в реальном времени, поэтому нужно заменить эту функцию на `PeekMessage()`, которая практически одинакова с `GetMessage()`. За исключением того, что `PeekMessage()` позволяет постоянно следить за очередью сообщений, и ее работа сводится к простым действиям: есть сообщения — обрабатываем, нет — ждем дальше. Прототип этой функции выглядит следующим образом:

```
BOOL PeekMessage(
    LPMSG    lpMsg,
    HWND     hwnd,
    UINT     wMsgFiltrenMin,
    UINT     wMsgFiltrenMax,
    UINT     wRemovwMsg);
```

Все отличия от `GetMessage()` заключаются в добавлении еще одного параметра `wRemovwMsg`, имеющего в своем распоряжении два флага:

- `PM_REMOVE` — при его использовании вызов функции `PeekMessage()` будет сопровождаться удалением сообщения из очереди;
- `PM_NOREMOVE` — оставляющий сообщение в очереди.

Затем конец функции `WinMain()` примет вид:

```
while (msg.message != WM_QUIT)
{
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    else
        RenderingDirect3D();
}
```

С помощью строки

```
msg.message != WM_QUIT
```

мы получаем возможность выхода из бесконечного цикла `while`.

Также добавьте в функцию `DeleteDirect3D()` код, освобождающий захваченные ресурсы по принципу "последним захватил — первым отдал".

```
if (pBufferVershin != NULL)
pBufferVershin -> Release();
```

Таким образом, весь код приведен в листинге 4.1.

Листинг 4.1. Файл `Treugolnik.cpp`

```
//-----
//  Treugolnik.cpp
//  рисуем треугольник
//-----
#include <windows.h>    // подключаем заголовочный файл Windows
#include <d3d9.h>        // подключаем заголовочный файл DirectX 9 SDK
//-----
//  глобальные переменные
//-----
LPDIRECT3D9             pDirect3D             = NULL;    // главный объект
```

```

LPDIRECT3DDEVICE9          pDirect3DDevice = NULL;      // устройство
LPDIRECT3DVERTEXBUFFER9    pBufferVerшин  = NULL;      // буфер вершин
struct CUSTOMVERTEX
{
    FLOAT x, y, z, rhw;      // координаты
    DWORD color;             // цвет вершин
};
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW|D3DFVF_DIFFUSE)
//-----
// функция
// InitialDirect3D()
// инициализация Direct3D
//-----
HRESULT InitialDirect3D(HWND hwnd)
{
    if (NULL == (pDirect3D = Direct3DCreate9(D3D_SDK_VERSION)))
        return E_FAIL;
    DISPLAYMODE Display;
    if (FAILED(pDirect3D -> GetAdapterDisplayMode(
        D3DADAPTER_DEFAULT, &Display)))
        return E_FAIL;
    D3DPRESENT_PARAMETERS Direct3DParametr;
    ZeroMemory(&Direct3DParametr, sizeof(Direct3DParametr));
    Direct3DParametr.Windowed = TRUE;
    Direct3DParametr.SwapEffect = D3DSWAPEFFECT_DISCARD;
    Direct3DParametr.BackBufferFormat = Display.Format;

    if (FAILED(pDirect3D -> CreateDevice(
        D3DADAPTER_DEFAULT,
        D3DDEVTYPE_HAL, hwnd,
        D3DCREATE_SOFTWARE_VERTEXPROCESSING,
        &Direct3DParametr, &pDirect3DDevice)))
        return E_FAIL;
    return S_OK;
}
//-----
// функция
// InitialBufferVerшин()
// инициализирует вершины

```

```
//-----
HRESULT InitialBufferVerшин()
{
    CUSTOMVERTEX Verшин[] =
    {
        {300.0f, 300.0f, 0.5f, 1.0f, 0x00000fff, }, // A
        {150.0f, 300.0f, 0.5f, 1.0f, 0x00000fff, }, // B
        {150.0f, 150.0f, 0.5f, 1.0f, 0x00000fff, }, // C
        // X      Y      Z      rhw      цвет
    };
    if (FAILED(pDirect3DDevice -> CreateVertexBuffer(
        3*sizeof(CUSTOMVERTEX), 0, D3DFVF_CUSTOMVERTEX,
        D3DPool_Default, &pBufferVerшин, NULL)))
        return E_FAIL;
    VOID* pBV;
    if (FAILED(pBufferVerшин -> Lock(0, sizeof(Verшин),
        (void**)&pBV, 0)))
        return E_FAIL;
    memcpy(pBV, Verшин, sizeof(Verшин)); // копирование
    pBufferVerшин -> Unlock(); // разблокирование
    return S_OK;
}

//-----
// функция
// RenderingDirect3D()
// рисование
//-----
VOID RenderingDirect3D()
{
    if (pDirect3DDevice == NULL) // проверяем ошибки
        return;
    pDirect3DDevice -> Clear(0, NULL, D3DCLEAR_TARGET,
        D3DCOLOR_XRGB(0, 255, 0), 1.0f, 0);
    pDirect3DDevice -> BeginScene(); // начало сцены
    // здесь происходит прорисовка сцены
    pDirect3DDevice -> SetStreamSource(0, pBufferVerшин, 0,
        sizeof(CUSTOMVERTEX));
    pDirect3DDevice -> SetFVF(D3DFVF_CUSTOMVERTEX);
    pDirect3DDevice -> DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);
}
```

```

    pDirect3DDevice -> EndScene();        // конец сцены
    pDirect3DDevice -> Present(NULL, NULL, NULL, NULL);
}

```

```

//-----

```

```

// функция
// DeleteDirect3D()
// освобождает захваченные ресурсы
//-----

```

```

VOID DeleteDirect3D()

```

```

{

```

```

    if (pBufferVershin != NULL)
        pBufferVershin -> Release();

```

```

    if (pDirect3DDevice != NULL)
        pDirect3DDevice -> Release();

```

```

    if (pDirect3D != NULL)
        pDirect3D -> Release();

```

```

}

```

```

//-----

```

```

// функция
// MainWinProc()
// здесь происходит обработка сообщений
//-----

```

```

LRESULT CALLBACK MainWinProc(HWND    hwnd,
                                UINT   msg,
                                WPARAM wParam,
                                LPARAM lParam)

```

```

{

```

```

    switch(msg)
    {

```

```

        case WM_DESTROY:
        {

```

```

            DeleteDirect3D();

```

```

            PostQuitMessage(0);

```

```

            return(0);

```

```

        }
    }

```

```

}

```

```

}

```

```
return DefWindowProc(hwnd, msg, wParam, lParam);
}
//-----
// функция
// WinMain
// входная точка приложения
//-----
int WINAPI WinMain(HINSTANCE hinstance,
                  HINSTANCE hprevinstance,
                  LPSTR lpcmdline,
                  int ncmdshow)
{
    WNDCLASSEX windowclass;    // создаем класс
    HWND hwnd;                // создаем дескриптор окна
    MSG msg;                   // идентификатор сообщения
    // определим класс окна WNDCLASSEX
    windowclass.cbSize = sizeof(WNDCLASSEX);
    windowclass.style = CS_DBLCLKS|CS_OWNDC|CS_HREDRAW|CS_VREDRAW;
    windowclass.lpfnWndProc = MainWinProc;
    windowclass.cbClsExtra = 0;
    windowclass.cbWndExtra = 0;
    windowclass.hInstance = hinstance;
    windowclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    windowclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    windowclass.hbrBackground = (HBRUSH)GetStockObject(GRAY_BRUSH);
    windowclass.lpszMenuName = NULL;
    windowclass.lpszClassName= "WINDOWSCCLASS";
    windowclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
    // регистрируем класс
    if (!RegisterClassEx(&windowclass))
        return(0);
    // теперь, когда класс зарегистрирован, можно создать окно
    if (!(hwnd = CreateWindowEx(
        NULL, "WINDOWSCCLASS", "Рисуем треугольник",
        WS_OVERLAPPEDWINDOW|WS_VISIBLE, 300, 150,
        500, 400, NULL, NULL, hinstance, NULL)))
        return(0);
    if (SUCCEEDED(InitialDirect3D(hwnd)))
    {
```

```

if (SUCCEEDED(InitialBufferVershin() ))
{
    ShowWindow(hwnd, SW_SHOWDEFAULT);
    UpdateWindow(hwnd );
    ZeroMemory(&msg, sizeof(msg));
    while(msg.message != WM_QUIT)
    {
        if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else
            RenderingDirect3D();
    }
}
return 0;
}

```

Рисуем квадрат

А сейчас давайте попробуем нарисовать квадрат. Мы знаем координаты нашего треугольника, поэтому просто дорисуем к гипотенузе созданного треугольника другой треугольник так, чтобы они соприкасались, и в итоге получился квадрат. Квадрат, образованный из двух треугольников, представлен на рис. 4.3.

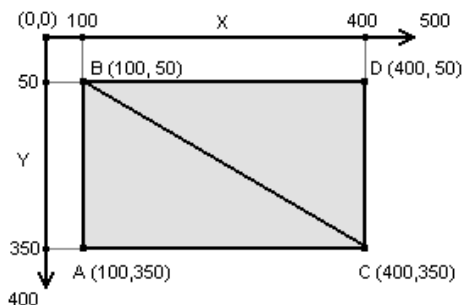


Рис. 4.3. Квадрат из треугольников

Координаты точек A , B и C нам известны, тогда из прямоугольника видно, что сторона AB равна стороне CD , а сторона BD равна стороне AC .

Зная координаты нашего нового треугольника, в функции инициализации вершин `InitialBufferVertexin()` структуры `CUSTOMVERTEX` `Vertexin` добавим координаты вершин нового треугольника. Но сначала, если хотите, чтобы предыдущий пример с треугольником у вас остался без изменения (см. листинг 4.1), можно создать новый файл под названием `Kvadrat.cpp`. Скопируйте код из файла `Treugolnik.cpp` и сохраните его под названием `Kvadrat.cpp`.

В структуру `Vertexin` (см. разд. "Создание буфера вершин" данной главы) добавляем новые координаты вершин:

```
CUSTOMVERTEX Vertexin[] =
{
    {300.0f, 300.0f, 0.5f, 1.0f, 0x00000fff}, // A
    {150.0f, 300.0f, 0.5f, 1.0f, 0x00000fff}, // B
    {150.0f, 150.0f, 0.5f, 1.0f, 0x00000fff}, // C

    {150.0f, 150.0f, 0.5f, 1.0f, 0x00000fff}, // C
    {300.0f, 150.0f, 0.5f, 1.0f, 0x00000fff}, // D
    {300.0f, 300.0f, 0.5f, 1.0f, 0x00000fff}, // A
    // X      Y      Z      rhw   цвет
};
```

В том месте, где создавался буфер вершин, с помощью функции `CreateVertexBuffer()`, помните, мы там указывали три вершины, умножая их на `sizeof(CUSTOMVERTEX)`, переправьте на число 6, оттого что у нас сейчас два треугольника, и, соответственно шесть вершин. И поскольку уже происходит прорисовка двух треугольников, необходимо в `RenderingDirect3D()` в функции `DrawPrimitive()` изменить параметр `PrimitiveCount` — счетчик количества выводимых примитивов — с 1 на 2 для двух треугольников. Весь остальной код остается без изменения. Откомпилируйте его и посмотрите, что получилось. Должен получиться синий квадрат на желтом фоне (рис. 4.4).

Теперь вы научились выводить двумерные фигуры на экран, а на уроке 5 мы разберемся, что такое матрицы, какие существуют функции для работы с матрицами в `Direct3D`, и уже на уроке 6 займемся рендерингом 3D-объектов.

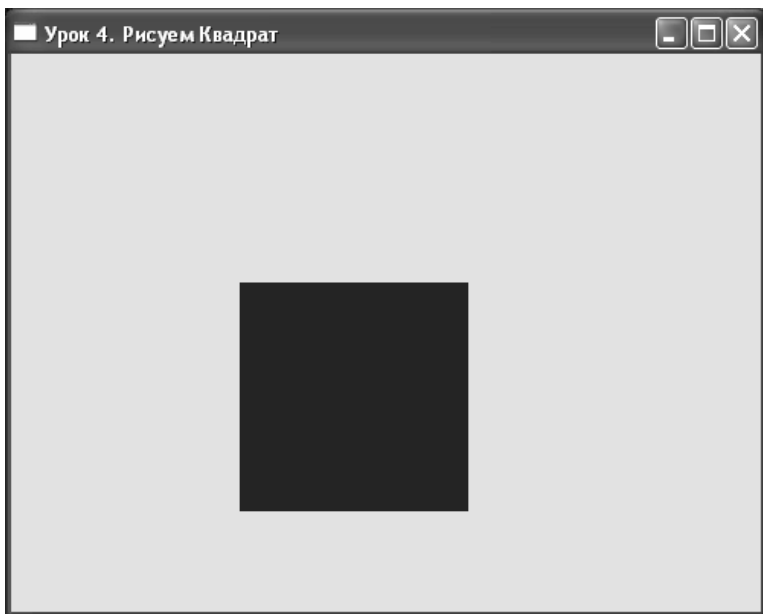


Рис. 4.4. Квадрат

Итоги урока

Этот урок объяснил вам подробно всю систему построения фигур при помощи буфера вершин, содержащего всевозможные данные о вершинах. Итогом урока послужило созданное вами окно с желтым фоном и синим треугольником в центре. Следующий урок объяснит роль матриц в преобразовании объекта, а также, если вы не были знакомы с матрицами, познакомит вас с элементарными понятиями.

Урок 5



Матрицы

Программирование трехмерной графики невозможно представить без использования матриц. Работа с матрицами строится на основе специальных функций Direct3D 9, но также имеются пути создания матричных преобразований вручную. В примерах, рассматриваемых в книге, используются функции Direct3D 9, но понимать, что такое матрицы, как происходит их сложение, умножение и другие операции, необходимо. Поэтому предстоит сделать небольшой "экскурс" в высшую математику, а потом уже поговорим о матрицах, используемых в Direct3D 9.

Итак, *матрица* — это определенный массив чисел с заранее известной размерностью строк и столбцов. Например:

$$\text{Матрица} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}.$$

Это обыкновенная матрица размерностью 2×2 , где цифры 1 и 2 — первая строка, а цифры 3 и 4 — вторая. Размерность матрицы может быть различной, например: 2×3 , 3×2 , 3×3 , 4×4 , 5×6 и т. д. Наверное, можно сделать ее и размером 3000×3000 , другое дело, как с ней работать, да и головных болей здесь не избежать.

Когда необходимо определить положение значения внутри матрицы, то говорят, что это значение находится в такой-то строке такого-то столбца. То есть некий элемент матрицы $m \times n$ находится в строке m столбца n .

Сложение и вычитание матриц

Сложение и вычитание матриц основано на арифметике начальных классов школы. Вы просто складываете или вычитаете две матрицы путем сложения или вычитания каждого элемента двух матриц, в результате получая итоговую матрицу. Единственным условием является размерность матриц, которая в обеих матрицах должна быть одинаковой.

Например, сложение осуществляется таким образом:

$$\begin{pmatrix} 3 & 7 \\ 4 & 8 \end{pmatrix} + \begin{pmatrix} 5 & 10 \\ 9 & 2 \end{pmatrix} = \begin{pmatrix} 8 & 17 \\ 13 & 10 \end{pmatrix},$$

а вычитание:

$$\begin{pmatrix} 6 & 9 \\ 1 & 9 \end{pmatrix} - \begin{pmatrix} 3 & 10 \\ 5 & 7 \end{pmatrix} = \begin{pmatrix} 3 & -1 \\ -4 & 2 \end{pmatrix}.$$

Умножение матриц

Умножение матриц бывает двух видов — *скалярным* и *матричным*. Скалярное умножение — это умножение всей матрицы на любое скалярное значение. Все элементы матрицы поочередно перемножаются на скалярное значение, а результат записывается в итоговую матрицу. Размерность матриц не имеет значения. Например:

$$9 \times \begin{pmatrix} 4 & 8 \\ 5 & 10 \end{pmatrix} = \begin{pmatrix} 36 & 72 \\ 45 & 90 \end{pmatrix}.$$

Умножение матриц между собой несколько отличается от скалярного произведения. Сама операция умножения матрицы A на матрицу B некоммутативна, т. е. $A \times B = C$, но $B \times A \neq C$. При умножении двух матриц обязательно должно соблюдаться такое условие: количество столбцов матрицы A равно количеству строк матрицы B . Согласно этому правилу и происходит умножение матриц, вы последовательно перемножаете каждое значение из первой строки матрицы A на каждое значение первого столбца матрицы B . В итоге они суммируются между собой, и получается результирующая матрица. Например:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{pmatrix} =$$

$$= \begin{pmatrix} A_{11} \times B_{11} + A_{12} \times B_{21} + A_{13} \times B_{31} & A_{11} \times B_{12} + A_{12} \times B_{22} + A_{13} \times B_{32} \\ A_{21} \times B_{11} + A_{22} \times B_{21} + A_{23} \times B_{31} & A_{21} \times B_{12} + A_{22} \times B_{22} + A_{23} \times B_{32} \end{pmatrix}.$$

Первое значение первой строки матрицы A умножается на первое значение первого столбца матрицы B . Далее, второе значение первой строки матрицы A умножается на второе значение первого столбца матрицы B . И так далее, до окончания значений в строке матрицы A и в столбце матрицы B . После чего все результаты суммируются и итоговое значение записывается в первую строку матрицы C слева направо. Размеры матриц могут быть любыми,

но количество столбцов первой матрицы должно быть равно количеству строк во второй матрице.

Единичная матрица

Единичная матрица — это матрица, в которой по главной диагонали слева направо располагаются единицы, а все остальные значения равны 0. Например:

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Эта матрица необходима для расчета преобразований. В данном случае единица — это просто некий показатель, умножение на который дает возможность произвести матричное преобразование.

Матрицы в Direct3D

Работа с графикой в Direct3D 9 основана на матричных операциях. Мы будем пользоваться специальными функциями из библиотеки утилит D3DX, входящей в состав Direct3D. Эта математическая библиотека предоставляет большие возможности по работе с матрицами, и ее необходимо подключить в ваш проект с помощью заголовочного файла d3dx9.h. Я вообще советую перед работой с каким-либо заголовочным файлом не лениться и изучить его код, он много о чем может рассказать.

Сам матричный подход в Direct3D весьма прост создав один раз объект, а именно заполнив буфер вершин данными, вы умножаете каждую вершину на ту или иную матрицу преобразования, получая при этом преобразованный объект. Сама же вершина представлена в виде вектора (X, Y, Z) в трехмерном пространстве. Все это делать вручную нет необходимости, в Direct3D определены матричные функции, использование которых весьма упрощает работу с матрицами. Также можно написать собственные матричные функции, если вас не устраивают стандартные методы работы.

В Direct3D определены три основные матрицы: мировая матрица (World Matrix), матрица вида (View Matrix) и матрица проекции (Projection Matrix). Сначала изучим краткие характеристики каждой из матриц, а потом разберем их по отдельности.

Мировая матрица — позволяет производить вращение, трансформацию и масштабирование объекта, а также наделяет каждый из объектов своей локальной системой координат.

Матрица вида — это, скажем так, ваши глаза или глаза пользователя. То есть она задает позицию и направление просмотра сцены.

Матрица проекции — создает проекцию трехмерного объекта на двумерный экран монитора. С ее помощью объект трансформируется, и начало координат переносится в переднюю часть рисуемого объекта, а также определяются передняя и задняя плоскости отсечения. Задавая последовательно значения для каждой из матриц, вы тем самым создаете трехмерную сцену, в которой получаете возможность перемещать, вращать, приближать, удалять и производить другие операции над объектами в зависимости от ваших требований.

Работая с матрицами в Direct3D, вершину можно задать четырьмя числами: X , Y , Z , W в виде матрицы, где W не равно 0. W — это коэффициент перспективы, необходимый для работы в трехмерном пространстве. В таком случае координаты вершин будут равны X/W , Y/W , Z/W и представляются в виде обыкновенной матрицы 4×4 .

Мировая матрица

Мировая матрица производит мировое преобразование системы координат для всех объектов, давая каждому из них свою локальную систему координат. Построение объекта происходит на основе заданной ему системы координат. На рис. 5.1 показано, как объект помещается "в мир" и получает свою систему координат.

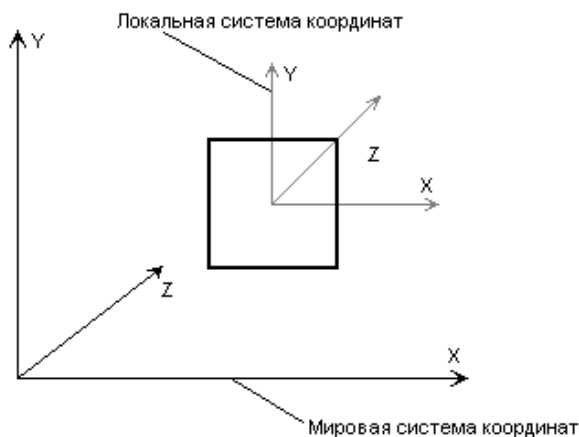


Рис. 5.1. Мировое преобразование

Мировое преобразование может состоять из любых комбинаций вращения, трансляции и масштабирования. Матрица вращения или циклического

сдвига позволяет вращать точку вокруг любой из осей координат. По оси X матрица вращения выглядит следующим образом:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\Theta & \sin\Theta & 0 \\ 0 & -\sin\Theta & \cos\Theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

где Θ — угол вращения в радианах.

Вращение вокруг оси Y :

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

Вращение вокруг оси Z :

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

Для этих целей в Direct3D есть функции `D3DXMatrixRotationX()`, `D3DXMatrixRotationY()`, `D3DXMatrixRotationZ()`, создающие соответственно вращение по осям X , Y , Z .

Матрица трансляции или перемещения перемещает точку с координатами (x, y, z) в новую точку (x_1, y_1, z_1) . Математически это выглядит следующим образом:

$$x_1 = x + T_x$$

$$y_1 = y + T_y$$

$$z_1 = z + T_z$$

Матрица трансляции имеет вид:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{pmatrix},$$

где T_x , T_y и T_z — значения смещения по осям X , Y и Z . Для удобства можно также воспользоваться функцией `D3DXMatrixTranslation()` из библиотеки утилит D3DX.

Матрица масштабирования — масштабирует точку с координатами (x, y, z) в точку с новыми значениями (x_1, y_1, z_1) . Чтобы произвести эту операцию нужно воспользоваться формулами:

$$x_1 = x \times S$$

$$y_1 = y \times S$$

$$z_1 = z \times S$$

Матрица масштабирования имеет общий вид:

$$\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

где S_x, S_y, S_z — значения коэффициентов растяжения или сжатия по осям X, Y, Z . `D3DXMatrixScale()` — функция в `Direct3D`, отвечающая за масштабирование.

Преимущество в использовании матриц состоит в том, что можно перемножить объект, а точнее вершины объекта на матрицы вращения, трансляции и масштабирования, получив при этом повернутый, перемещенный и масштабированный объект. Или, например, сначала перемножить между собой все необходимые матрицы, получив в итоге какую-то исходную матрицу, а потом умножить вершины объекта на полученную исходную матрицу. Перемножение матриц между собой носит название *матричной конкатенации* и выражается в `Direct3D` функцией `D3DXMatrixMultiply()`.

Матрица вида

Матрица вида определяет местоположение камеры просмотра сцены и может состоять из любых комбинаций трансляции и вращения. Сама камера находится в начале системы координат и направлена в положительную сторону оси Z , а матрица вида перемещает объект в мире вокруг позиции камеры. С помощью функции `D3DXMatrixLookAtLH(Out, eye, at, up)` в `Direct3D` создается матрица вида, где `Out` — это итоговая матрица всей операции, а значения `eye`, `at` и `up` задаются при помощи структуры `D3DXVECTOR3`, описывающей точку в трехмерном пространстве посредством задания трех координат по осям X, Y, Z . Например,

```
D3DXVECTOR3 eye(4, 3, 5);
```

Значение `eye` показывает точку нахождения камеры, значение `at` — это то место, куда камера направлена, и значение `up` — то, что в итоге видим. Обычно значение `up` задается таким образом:

```
D3DXVECTOR3 up(0, 1, 0);
```

В Direct3D используется левосторонняя система координат, показанная на рис. 5.2.

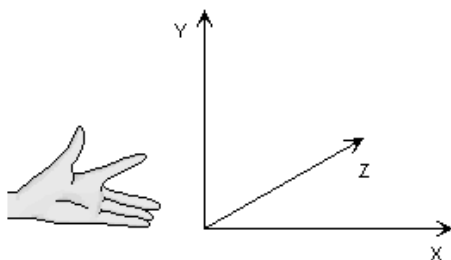


Рис. 5.2. Левосторонняя система координат

Поэтому применяется функция `D3DXMatrixLookAtLH()`. В том случае, если каким-то образом используется правосторонняя система координат, необходимо применить функцию `D3DXMatrixLookAtRH()`, предназначенную для этих целей.

Матрица проекции

Матрица проекции довольно сложная, вкратце весь ее смысл заключается в правильном проецировании трехмерного объекта в двумерной плоскости проекции. Объем видимости объекта регулируется передней и задней областями отсечения. На рис. 5.3 это хорошо видно.

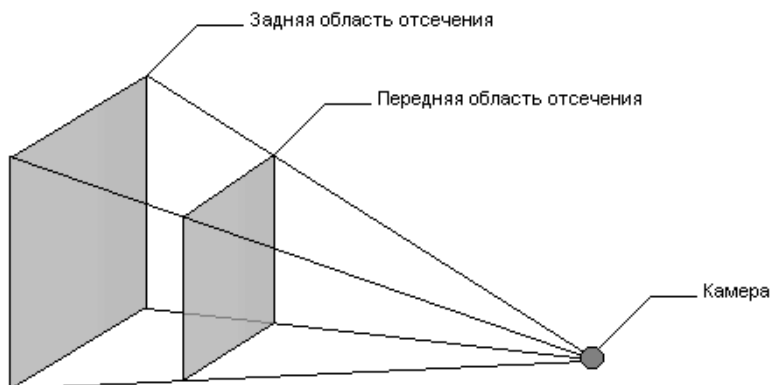


Рис. 5.3. Передняя и задняя области отсечения

Также с помощью матрицы проекции происходит перенос системы координат в переднюю область отсечения, как показано на рис. 5.4. При этом для расчета перспективы преобразования координаты по осям X и Y делятся на Z .

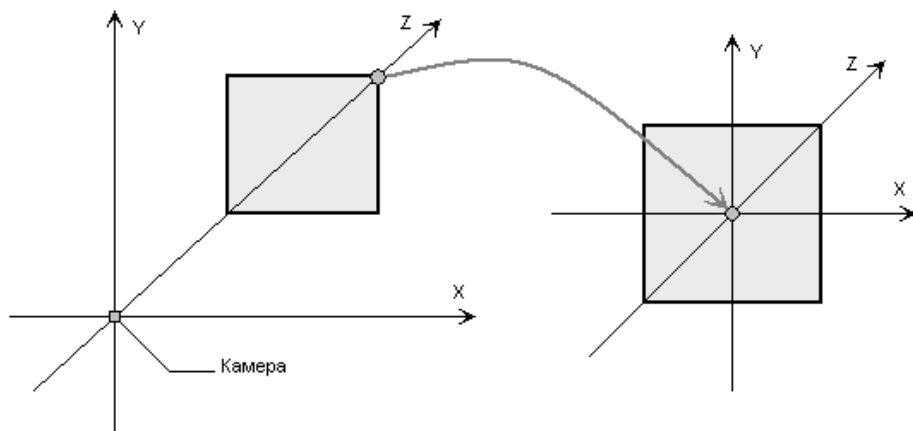


Рис. 5.4. Преобразование матрицы проекции

Что касается матричных структур в Direct3D, то они представлены несколькими видами. Рассмотрим структуры, которые используются в книге. Первая структура `D3DMATRIX` имеет вид матрицы 4×4 , ее прототип выглядит следующим образом:

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

        };
        float m. [4] [4];
    };
} D3DMATRIX;
```

Другая, более мощная структура — `D3DXMATRIX` из библиотеки утилит `D3DX`, в основном мы будем пользоваться ею.

Прототип данной структуры описывается так:

```
typedef struct D3DXMATRIX : public D3DMATRIX {
public:
    D3DXMATRIX() {} ;
    D3DXMATRIX(CONST FLOAT *);
    D3DXMATRIX(CONST D3DMATRIX&);
    D3DXMATRIX(FLOAT _11, FLOAT _12, FLOAT _13, FLOAT _14,
                FLOAT _21, FLOAT _22, FLOAT _23, FLOAT _24,
                FLOAT _31, FLOAT _32, FLOAT _33, FLOAT _34,
                FLOAT _41, FLOAT _42, FLOAT _43, FLOAT _44);

    FLOAT& operator() (UINT Row, UINT Col);
    FLOAT operator() (UINT Row, UINT Col) const;

    operator FLOAT*();
    operator CONST FLOAT*() const;

    D3DXMATRIX& operator *= (CONST D3DXMATRIX&);
    D3DXMATRIX& operator += (CONST D3DXMATRIX&);
    D3DXMATRIX& operator -= (CONST D3DXMATRIX&);
    D3DXMATRIX& operator *= (FLOAT);
    D3DXMATRIX& operator /= (FLOAT);

    D3DXMATRIX operator + () const;
    D3DXMATRIX operator - () const;

    D3DXMATRIX operator * (CONST D3DXMATRIX&) const;
    D3DXMATRIX operator + (CONST D3DXMATRIX&) const;
    D3DXMATRIX operator - (CONST D3DXMATRIX&) const;
    D3DXMATRIX operator * (FLOAT) const;
    D3DXMATRIX operator / (FLOAT) const;

    friend D3DXMATRIX operator * (FLOAT, CONST D3DXMATRIX&);

    BOOL operator == (CONST D3DXMATRIX&) const;
    BOOL operator != (CONST D3DXMATRIX&) const;

} D3DXMATRIX, *LPD3DXMATRIX;
```

И еще одна матрица D3DXMATRIXA16 описывается следующим образом:

```
typedef struct _D3DXMATRIXA16 : public D3DMATRIX
{
    _D3DXMATRIXA16();
    _D3DXMATRIXA16(CONST FLOAT * f);
    _D3DXMATRIXA16(CONST D3DMATRIX& m);
    _D3DXMATRIXA16(FLOAT _11, FLOAT _12, FLOAT _13, FLOAT _14,
                    FLOAT _21, FLOAT _22, FLOAT _23, FLOAT _24,
                    FLOAT _31, FLOAT _32, FLOAT _33, FLOAT _34,
                    FLOAT _41, FLOAT _42, FLOAT _43, FLOAT _44);

    void* operator new(size_t s);
    void* operator new[] (size_t s);

    void operator delete(void* p);
    void operator delete[] (void* p);

    struct _D3DXMATRIXA16& operator=(CONST D3DXMATRIX& rhs);
} _D3DXMATRIXA16;
```

Итоги урока

На этом уроке были рассмотрены сложение, вычитание и умножение матриц, а также умножение матрицы на некую скалярную величину. Также мы изучили принципы матричных преобразований и узнали, что имеются мировая матрица, матрица вида и матрица проекции, включающие в себя ряд разнообразных матриц. На *уроке 6* мы воспользуемся этими знаниями, перейдем в трехмерную систему координат и построим настоящий трехмерный объект.

Урок 6



Вывод на экран 3D-объекта

На этом уроке мы перейдем в трехмерную систему координат, где вначале нарисуем плоский квадрат и будем его вращать по оси X , а потом создадим куб, используя для этого индексацию вершин.

Создайте новый проект Urok6, добавьте новый пустой файл под названием MatrixKvadrat.cpp и скопируйте код Kvadrat.cpp из *урока 4*, находящийся на компакт-диске в папке Code\Urok4. Теперь мы будем пользоваться матричными функциями, поэтому необходимо добавить заголовочный файл:

```
#include <d3dx9.h>
```

Этот заголовочный файл подключает к нашему проекту вспомогательную библиотеку утилит для Direct3D, в которой находятся функции и структуры для работы с матрицами. Также в сам проект подключается одноименная библиотека d3dx9.lib. В том месте, где на *уроке 3* была подключена библиотека d3d9.lib, через пробел добавьте новую библиотеку. И еще нам будет необходим следующий заголовочный файл:

```
#include <mmsystem.h>
```

Это стандартный заголовочный файл для работы с системными функциями. Мы будем пользоваться функцией timeGetTime() для работы со временем. Все вышеперечисленные файлы добавьте в начало файла MatrixKvadrat.cpp. Изменение коснется и структуры CUSTOMVERTEX, с помощью которой определяется формат вершин. Ее вид теперь будет следующий:

```
Struct CUSTOMVERTEX
{
    FLOAT X, Y, Z;    // координаты
    DWORD color;      // цвет вершин
};

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE)
```

На *уроке 4* применялся формат вершин FVF для преобразованной вершины D3DFVF_XYZRHW. Тогда у нас был двумерный случай, и мы обязаны были преобразовывать вершины в эту плоскость, посредством необходимой записи.

Сейчас мы пользуемся 3D-координатами, и вершины останутся непреобразованными, поскольку мы будем использовать матрицы для работы по преобразованию и представлению нашего 3D-объекта в пространстве. Флаг `D3DFVF_XYZ` оставляет вершины непреобразованными.

Далее идет функция `InitialDirect3D()`, код которой тоже почти не претерпел никаких изменений. В конце всей функции добавятся две строки:

```
// отключить отсечение Direct3D
pDirect3DDevice -> SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE);

// отключить освещение Direct3D
pDirect3DDevice -> SetRenderState(D3DRS_LIGHTING, FALSE);
```

В данных строках вызывается функция `IDirect3DDevice9::SetRenderState`, с помощью которой происходит установка режима отображения для устройства Direct3D. В первом случае отключается режим отсечения, для того, чтобы при вращении 3D-объекта можно было видеть все его стороны, а во втором — запрещается работа со светом. Прототип функции `SetRenderState()` выглядит следующим образом:

```
HRESULT SetRenderState(
    D3DRENDERSTATETYPE State,
    DWORD                Value);
```

Функция `SetRenderState()` имеет такие параметры:

- `State` — значением для этой переменной может быть любой из членов перечисляемого типа `D3DRENDERSTATETYPE`, обозначающего тип работы для устройства `pDirect3DDevice`;
- `Value` — это значение полностью зависит от параметра `State` и также определяет, как параметр `State` будет работать.

В первом моменте вызов функции происходит со значением `D3DRS_CULLMODE`, определяющим режим отсечения для сцены, а второе значение — `D3DCULL_NONE` — запрещает отсечение в данном примере.

Следующий вызов функции `IDirect3DDevice9::SetRenderState` сопровождается значением `D3DRS_LIGHTING` — освещение, и значением `FALSE` — для его отключения. Чтобы включить свет, ставится значение `TRUE`, но поскольку мы пока не используем освещение, а закрашиваем вершины в синий цвет, то поставим значение `FALSE`.

Следующая функция в нашем коде — `InitialBufferVerшин()`. Эта функция инициализирует вершины, потом создает буфер вершин, блокирует его для записи, копирует в него данные и разблокирует. Единственное, что меняется в этой функции, — это инициализация вершин. Если предыдущая инициализация была ориентирована на двумерную систему координат, где

применялись преобразованные вершины и все данные записывались исходя из общей системы координат, то сейчас, когда мы имеем дело с матричным преобразованием, все в корне изменилось. С помощью матриц происходит преобразование, в котором появляются так называемые локальные координаты для 3D-объекта, в отношении которых и происходит его построение. Direct3D использует левостороннюю систему координат, где ось X идет слева направо по нижней кромке дисплея, Y — снизу вверх по левой стороне, а ось Z удаляется от наблюдателя. Поэтому данные для квадрата необходимо изменить таким образом:

```
CUSTOMVERTEX Vershin[] = {
    {1.0f, -1.0f, 0.0f, 0x00000fff},    // A
    {-1.0f, -1.0f, 0.0f, 0xff000fff},   // B
    {-1.0f, 1.0f, 0.0f, 0x00000fff},    // C
    {1.0f, 1.0f, 0.0f, 0xff000fff},     // D
    {1.0f, -1.0f, 0.0f, 0x00000fff},    // A
};

// X      Y      Z      цвет
```

И посмотрите на рис. 6.1, на котором видно, как меняется система координат, а также сам квадрат, имеющий уже другие координаты.

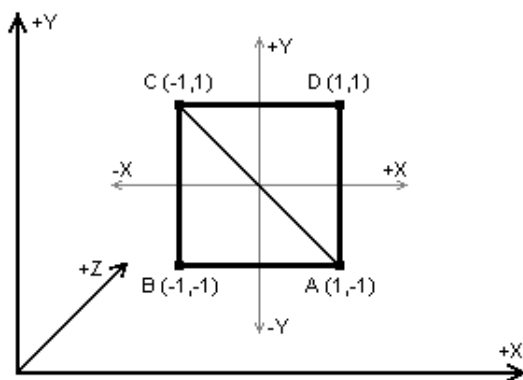


Рис. 6.1. Смена системы координат

Наш квадрат по-прежнему состоит из двух треугольников, координаты которых задаются от 1 до -1 относительно своей собственной системы координат.

Матрицы преобразования

Пойдем дальше. В коде `MatrixKvadrat.cpp` у нас появится новая функция `Matrix()`, в которой будут происходить все матричные преобразования. Текст функции выглядит следующим образом:

```
VOID Matrix()
{
    D3DXMATRIX MatrixWorld;        // мировая матрица
    D3DXMATRIX MatrixView;         // матрица вида
    D3DXMATRIX MatrixProjection;   // матрица проекции
    // MatrixWorld
    UINT Time = timeGetTime() % 5000;
    FLOAT Angle = Time * (2.0f * D3DX_PI) / 5000.0f;
    D3DXMatrixRotationX(&MatrixWorld, Angle);
    pDirect3DDevice -> SetTransform(D3DTS_WORLD, &MatrixWorld);
    // MatrixView
    D3DXMatrixLookAtLH(&MatrixView, &D3DXVECTOR3(0.0f, 0.0f, -8.0f),
                      &D3DXVECTOR3(0.0f, 0.0f, 0.0f),
                      &D3DXVECTOR3(0.0f, 1.0f, 0.0f));
    pDirect3DDevice -> SetTransform(D3DTS_VIEW, &MatrixView);
    // MatrixProjection
    D3DXMatrixPerspectiveFovLH(&MatrixProjection, D3DX_PI/4,
                              1.0f, 1.0f, 100.0f);
    pDirect3DDevice -> SetTransform(D3DTS_PROJECTION, &MatrixProjection);
}
```

В первых трех строках функции `Matrix()` создаются три объекта: мировая матрица (`MatrixWorld`), матрица вида (`MatrixView`) и матрица проекции (`MatrixProjection`). Следующие четыре строки относятся к установке мировой матрицы, с помощью которой будет вращаться наш 3D-объект.

С помощью системной функции `timeGetTime()` мы берем время, прошедшее с начала работы системы Windows, и делим по модулю на 5000, сохранив полученный результат в переменной `Time`:

```
UINT Time = timeGetTime() % 5000;
```

Все значения измеряются в миллисекундах. Дальше в строке

```
FLOAT Angle = Time * (2.0f * D3DX_PI) / 5000.0f;
```

мы вычисляем угол вращения в радианах. Здесь, я думаю, все понятно, замечу только, что значение `D3DX_PI` — ни что иное, как число $\pi \approx 3.14$, определенное в библиотеке утилит `Direct3D` как константа.

Следующая строка кода вызывает функцию `D3DXMatrixRotationX()`, с помощью которой формируется матрица для вращения вокруг оси *X*. Таких функций, как вы уже знаете, всего три — еще существуют `D3DXMatrixRotationY()` и `D3DXMatrixRotationZ()`. Все прототипы этих функций одинаковы, поэтому рассмотрим функцию, используемую в этом примере:

```
D3DXMATRIX *D3DXMatrixRotationX(  
    D3DXMATRIX* pOut,  
    FLOAT       Angle);
```

Функция имеет следующие параметры:

- ❑ `pOut` — указатель на структуру `D3DXMATRIX`, в которую помещается исходный результат операций;
- ❑ `Angle` — угол вращения в радианах по оси *X*. Попробуйте поставить 0 и посмотрите, что получится.

После компиляции кода, разобранный на этом уроке, поэкспериментируйте с этими тремя функциями при различных значениях угла вращения.

Последний шаг в установке мировой матрицы заключается в вызове функции `IDirect3DDevice9::SetTransform`, применяемой для всех преобразований объекта. Забегая вперед, скажу, что вызовом этой функции заканчиваются все преобразования для любых матриц. Название самой функции `SetTransform()` подразумевает дословно установку преобразования. То есть, осуществив преобразования, необходимо вызвать данную функцию, чтобы эти преобразования вступили в силу:

```
pDirect3DDevice -> SetTransform(D3DTS_WORLD, &MatrixWorld);
```

Рассмотрим прототип функции `SetTransform()`:

```
HRESULT SetTransform(  
    D3DTRANSFORMSTATETYPE State,  
    CONST D3DMATRIX*      pMatrix);
```

Функция имеет следующие параметры:

- ❑ `State` — этот параметр указывает на происходящее преобразование. В данном случае оно происходит для мировой матрицы — это `D3DTS_WORLD`;
- ❑ `pMatrix` — указатель на матрицу, установленную как текущее преобразование.

Следующая на очереди матрица вида — своего рода камера, показывающая, в каком месте располагаются наши глаза. То, на что смотрим (середина всей области просмотра) и что видим, т. е. направление взгляда. Вызов функции `D3DXMatrixLookAtLH()` является определяющим в этом вопросе, ее прототип выглядит следующим образом:

```
D3DXMATRIX *D3DXMatrixLookAtLH(  
    D3DXMATRIX* pOut,
```

```
CONST D3DXVECTOR3* pEye,
CONST D3DXVECTOR3* pAt,
CONST D3DXVECTOR3* pUp);
```

Функция `D3DXMatrixLookAtLH()` имеет такие параметры:

- ❑ `pOut` — результат выполнения этой функции. В качестве указателя на структуру `D3DXMATRIX` выступает `MatrixView`, наша матрица вида;
- ❑ `pEye` — указатель на структуру `D3DXVECTOR3`, который определяет точку, из которой происходит просмотр сцены, т. е. место, где мы находимся;
- ❑ `pAt` — указатель на структуру `D3DXVECTOR3`, определяющий то, на что мы смотрим;
- ❑ `pUp` — указатель на структуру `D3DXVECTOR3`, определяющий то, что мы видим. Обычно это точка с координатами (0,1,0).

С первым параметром все понятно — это результат выполнения всей функции. Три оставшиеся параметра — указатели на структуру `D3DXVECTOR3`, описывающие вектор в трехмерном пространстве при помощи трех вещественных чисел с заданными координатами по осям X , Y и Z . Сама структура выглядит таким образом:

```
typedef struct D3DXVECTOR3 : public D3DVECTOR{
public:
    D3DXVECTOR3() {} ;
    D3DXVECTOR3(CONST FLOAT *);
    D3DXVECTOR3(CONST D3DVECTOR&);
    D3DXVECTOR3(FLOAT X, FLOAT Y, FLOAT Z);

    operator FLOAT*();
    operator CONST FLOAT*() const;

    // операторы назначения
    D3DXVECTOR3& operator += (CONST D3DXVECTOR3&);
    D3DXVECTOR3& operator -= (CONST D3DXVECTOR3&);
    D3DXVECTOR3& operator *= (FLOAT);
    D3DXVECTOR3& operator /= (FLOAT);

    // одиночные операторы
    D3DXVECTOR3 operator + () const;
    D3DXVECTOR3 operator - () const;

    // двойные операторы
    D3DXVECTOR3 operator + (CONST D3DXVECTOR3&) const;
```

```

D3DXVECTOR3 operator - (CONST D3DXVECTOR3&) const;
D3DXVECTOR3 operator * (FLOAT) const;
D3DXVECTOR3 operator / (FLOAT) const;

friend D3DXVECTOR3 operator * (FLOAT, CONST struct D3DXVECTOR3&);

BOOL operator == (CONST D3DXVECTOR3&) const;
BOOL operator != (CONST D3DXVECTOR3&) const;

} D3DXVECTOR3, *LPD3DXVECTOR3;

```

Довольно сложная структура. Когда вы ставите определяющее значение для `pEye` — `D3DXVECTOR3 (0.0f, 0.0f, -8.0f)`, как у нас в примере, то вы задаете точку, откуда происходит просмотр 3D-объекта, с помощью трех значений X , Y , Z . Задав $Z = -8$ модулей, мы отдалим от себя объект на это значение, а два нуля для значений X и Y указывают, что мы смотрим в центр объекта. Обязательно поэкспериментируйте со значениями для `pAt`, `pEye` и `pUp`.

За вызовом этой функции сразу же следует вызов `SetTransform()` для вступления в силу всех произошедших преобразований:

```
pDirect3DDevice -> SetTransform(D3DTS_VIEW, &MatrixView)
```

Параметры функции `SetTransform()` означают следующее:

- ☐ `D3DTS_VIEW` — указывает на преобразование, установленное для матрицы вида;
- ☐ `MatrixView` — матрица, установленная как текущее преобразование.

И последний шаг — установка матрицы проекции для отображения трехмерной сцены на двумерной плоскости монитора.

В примере сначала вызывается функция `D3DXMatrixPerspectiveFovLH()` для установки матрицы проекции. Ее прототип записывается так:

```

D3DXMATRIX *D3DXMatrixPerspectiveFovLH(
    D3DXMATRIX* pOut,
    FLOAT       fovY,
    FLOAT       Aspect,
    FLOAT       zn,
    FLOAT       zf);

```

Функция `D3DXMatrixPerspectiveFovLH()` имеет следующие параметры:

- ☐ `pOut` — результат всей операции, указатель на структуру `D3DXMATRIX` — это `MatrixProjection`;

- fovY — определяет поле зрения в направлении оси Y, измеряется в радианах. Типичное поле — $D3DX_PI/4$;
- Aspect — коэффициент сжатия, используемый для соотношения геометрических размеров. Обычно значение равно единице;
- zn — передний план отсечения сцены;
- zf — задний план отсечения сцены. За границей двух параметров zn и zf ничего не рисуется.

Чтобы текущие преобразования вступили в силу, вызываем функцию `SetTransform()`:

```
pDirect3DDevice -> SetTransform(D3DTS_PROJECTION, &MatrixView);
```

где используемые параметры означают следующее:

- D3DTS_PROJECTION — преобразование, установленное для матрицы проекции;
- MatrixView — матрица, установленная как текущее преобразование.

После того как мы создали функцию `Matrix()`, можно поместить ее в `RenderingDirect3D()`, в место, где происходит прорисовка сцены. Также эту функцию можно поместить в любое место, необходимое вам, главное, чтобы она влияла непосредственно на объект, рисуемый на экране. Весь остальной код остается без изменений. Все, что у нас получилось, приведено в листинге 6.1.

Листинг 6.1. Файл `MatrixKvadrat.cpp`

```
//-----
//  MatrixKvadrat.cpp
//  рисуем квадрат
//-----
#include <windows.h> // подключаем заголовочный файл Windows
#include <d3d9.h>     // подключаем заголовочный файл DirectX 9 SDK
#include <d3dx9.h>    // подключаем заголовочный файл из D3DX-утилит
                      // для работы с матрицами
#include <mmsystem.h> // подключаем системную библиотеку
//-----
//  глобальные переменные
//-----
LPDIRECT3D9 pDirect3D = NULL;           // главный объект в Direct3D
LPDIRECT3DDEVICE9 pDirect3DDevice = NULL; // устройство
LPDIRECT3DVERTEXBUFFER9 pBufferVershin = NULL; // буфер вершин
struct CUSTOMVERTEX
```

```

{
    FLOAT X, Y, Z;    // координаты точки
    DWORD color;      // цвет вершины
};

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE) // формат вершины
//-----
// функция
// IDirect3D()
// инициализация IDirect3D
//-----
HRESULT IDirect3D(HWND hwnd)
{
    if (NULL == (pDirect3D = Direct3DCreate9(D3D_SDK_VERSION)))
        return E_FAIL;
    D3DDISPLAYMODE Display;
    if (FAILED(pDirect3D -> GetAdapterDisplayMode(D3DADAPTER_DEFAULT,
                                                    &Display)))
        return E_FAIL;
    D3DPRESENT_PARAMETERS Direct3DParametr;
    ZeroMemory(&Direct3DParametr, sizeof(Direct3DParametr));
    Direct3DParametr.Windowed = TRUE;
    Direct3DParametr.SwapEffect = D3DSWAPEFFECT_DISCARD;
    Direct3DParametr.BackBufferFormat = Display.Format;
    if (FAILED(pDirect3D -> CreateDevice(D3DADAPTER_DEFAULT,
                                         D3DDEVTYPE_HAL, hwnd,
                                         D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                         &Direct3DParametr, &pDirect3DDevice)))
        return E_FAIL;
    pDirect3DDevice -> SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE);
    // отключение освещения IDirect3D
    pDirect3DDevice -> SetRenderState(D3DRS_LIGHTING, FALSE);
    return S_OK;
}
//-----
// функция
// IDirect3DDevice()
// инициализирует вершины
//-----
HRESULT IDirect3DDevice()

```

```

{
    CUSTOMVERTEX Vershin[] =
    {
        {1.0f, -1.0f, 0.0f, 0x00000fff},    // A
        {-1.0f, -1.0f, 0.0f, 0xff000fff},  // B
        {-1.0f, 1.0f, 0.0f, 0x00000fff},   // C

        {-1.0f, 1.0f, 0.0f, 0x00000fff},   // C
        {1.0f, 1.0f, 0.0f, 0xff000fff},     // D
        {1.0f, -1.0f, 0.0f, 0x00000fff},    // A
        // X      Y      Z      цвет
    };
    if (FAILED(pDirect3DDevice -> CreateVertexBuffer(
        6*sizeof(CUSTOMVERTEX), 0, D3DFVF_CUSTOMVERTEX,
        D3DPOOL_DEFAULT, &pBufferVershin, NULL)))
        return E_FAIL;
    // блокирование
    VOID* pBV;
    if (FAILED pBufferVershin -> Lock(0, sizeof(Vershin), (void**)&pBV, 0))
        return E_FAIL;
    memcpy(pBV, Vershin, sizeof(Vershin)); // копирование
    pBufferVershin -> Unlock();           // разблокирование
    return S_OK;
}

//-----
// функция
// Matrix()
// мировая матрица, матрица вида, матрица проекции
//-----
VOID Matrix()
{
    D3DXMATRIX MatrixWorld;    // мировая матрица
    D3DXMATRIX MatrixView;     // матрица вида
    D3DXMATRIX MatrixProjection; // матрица проекции
    // MatrixWorld
    UINT Time = timeGetTime() % 5000;
    FLOAT Angle = Time * (2.0f * D3DX_PI) / 5000.0f;
    D3DXMatrixRotationY(&MatrixWorld, Angle);
    pDirect3DDevice -> SetTransform(D3DTS_WORLD, &MatrixWorld);
}

```

```

// MatrixView
D3DXMatrixLookAtLH(&MatrixView, &D3DXVECTOR3(0.0f, 0.0f, -8.0f),
                  &D3DXVECTOR3(0.0f, 0.0f, 0.0f),
                  &D3DXVECTOR3(0.0f, 1.0f, 0.0f));
pDirect3DDevice -> SetTransform(D3DTS_VIEW, &MatrixView);
// MatrixProjection
D3DXMatrixPerspectiveFovLH(&MatrixProjection,
                          D3DX_PI/4, 1.0f, 1.0f, 100.0f);
pDirect3DDevice -> SetTransform(D3DTS_PROJECTION, &MatrixProjection);
}
//-----
// функция
// RenderingDirect3D()
// рисование
//-----
VOID RenderingDirect3D()
{
    if (pDirect3DDevice == NULL)        // проверка ошибок
        return;
    pDirect3DDevice -> Clear(0, NULL, D3DCLEAR_TARGET,
                          D3DCOLOR_XRGB(255, 255, 50), 1.0f, 0);
    pDirect3DDevice -> BeginScene();    // начало сцены
    Matrix();
    // здесь происходит прорисовка сцены
    pDirect3DDevice -> SetStreamSource(0, pBufferVershin, 0,
                                      sizeof(CUSTOMVERTEX));
    pDirect3DDevice -> SetFVF(D3DFVF_CUSTOMVERTEX);
    pDirect3DDevice -> DrawPrimitive(D3DPT_TRIANGLELIST, 0, 2);
    pDirect3DDevice -> EndScene();      // конец сцены
    pDirect3DDevice -> Present(NULL, NULL, NULL, NULL);
}
//-----
// функция
// DeleteDirect3D()
// освобождает захваченные ресурсы
//-----
VOID DeleteDirect3D()
{

```

```

    if (pBufferVershin != NULL)
        pBufferVershin -> Release();

    if (pDirect3DDevice != NULL)
        pDirect3DDevice -> Release();

    if (pDirect3D != NULL)
        pDirect3D -> Release();
}

//-----
//  функция
//  MainWinProc()
//  здесь происходит обработка сообщений
//-----

LRESULT CALLBACK MainWinProc(HWND      hwnd,
                               UINT      msg,
                               WPARAM     wparam,
                               LPARAM     lparam)
{
    switch(msg)
    {
        case WM_DESTROY:
        {
            DeleteDirect3D();
            PostQuitMessage(0);
            return(0);
        }
    }
    return DefWindowProc(hwnd, msg, wparam, lparam);
}

//-----
//  функция
//  WinMain
//  входная точка приложения
//-----

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

```

```
{
    WNDCLASSEX windowclass;    // создаем класс
    HWND        hwnd;          // создаем дескриптор окна
    MSG          msg;           // идентификатор сообщения
    // определим класс окна WNDCLASSEX
    windowclass.cbSize = sizeof(WNDCLASSEX);
    windowclass.style = CS_DBLCLKS|CS_OWNDC|CS_HREDRAW|CS_VREDRAW;
    windowclass.lpfnWndProc = MainWinProc;
    windowclass.cbClsExtra = 0;
    windowclass.cbWndExtra = 0;
    windowclass.hInstance = hinstance;
    windowclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    windowclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    windowclass.hbrBackground= (HBRUSH)GetStockObject(GRAY_BRUSH);
    windowclass.lpszMenuName = NULL;
    windowclass.lpszClassName = "WINDOWSCCLASS";
    windowclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
    // регистрируем класс
    if (!RegisterClassEx(&windowclass))
        return(0);
    // теперь, когда класс зарегистрирован, можно создать окно
    if (! (hwnd = CreateWindowEx(NULL, "WINDOWSCCLASS", "Вращаем Квадрат",
                                WS_OVERLAPPEDWINDOW|WS_VISIBLE,
                                300, 150, 500, 400, NULL, NULL,
                                hinstance, NULL)))
        return 0;
    if (SUCCEEDED(InitialDirect3D(hwnd)))
    {
        if (SUCCEEDED(InitialBufferVershin() ))
        {
            ShowWindow(hwnd, SW_SHOWDEFAULT);
            UpdateWindow(hwnd);
            ZeroMemory(&msg, sizeof(msg));
            while(msg.message != WM_QUIT)
            {
                if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
                {
                    TranslateMessage(&msg);
                    DispatchMessage(&msg);
                }
            }
        }
    }
}
```

```

    }
    else
        RenderingDirect3D();
    }
}
}
return 0;
}

```

После успешной компиляции вы увидите синий квадрат, вращающийся по оси X . Как ранее уже упоминалось, с помощью трех функций — `D3DMatrixRotationX()`, `D3DMatrixRotationY()` и `D3DMatrixRotationZ()` — можно вращать объект по трем осям координат.

Рисуем куб

Давайте сделаем из квадрата (см. листинг 6.1) полноценный 3D-объект — куб. Преобразуем нашу структуру `Vershin`, добавив в нее недостающие данные для инициализации всех вершин куба. Но сначала немного теории. Как я уже замечал, для этих целей можно использовать концепцию индексации вершин, но перед тем как воспользоваться ею, посмотрим, как можно сделать то же самое без индексации вершин. Ряд последовательных рисунков поможет нам в этом. По идее на первом рисунке из этой серии должен быть изображен квадрат из первой части урока (см. разд. "Матрицы преобразования"), но чтобы не дублировать пройденный материал, мы его пропустим. Однако вы должны держать эту сторону куба в уме — она у нас уже готовая. Далее будем следовать по часовой стрелке вокруг оси Y .

Итак, строим левую боковую сторону куба, состоящую из двух треугольников. Смотрим на рис. 6.2.

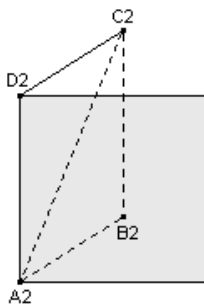


Рис. 6.2. Левая боковая сторона куба

Координаты первой вершины A_2 : $X = -1$, $Y = -1$, $Z = 1$, здесь все понятно. Следующая вершина B_2 , ее координаты идут уже по оси Z . Поскольку Z имеет значение -1 , то $X = -1$ и $Y = -1$, и эти координаты будут строиться уже относительно оси Z , со знаком минус. В итоге, координаты вершин для первого треугольника: $A_2 = (-1, -1, 1)$, $B_2 = (-1, -1, -1)$, $C_2 = (-1, 1, -1)$.

А для второго: $C_2 = (-1, 1, -1)$, $D_2 = (1, 1, -1)$, $A_2 = (-1, -1, -1)$.

Имея две стороны куба, нарисуем заднюю стенку, которая будет строиться от координаты $Z = -1$. Смотрим на рис. 6.3.

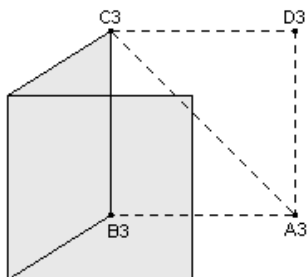


Рис. 6.3. Задняя сторона куба

Соответственно, координаты вершин для первого треугольника будут такими: $A_3 = (1, -1, -1)$, $B_3 = (-1, -1, -1)$, $C_3 = (-1, 1, -1)$.

А для второго: $C_3 = (-1, 1, -1)$, $D_3 = (1, 1, -1)$, $A_3 = (1, -1, -1)$.

Надеюсь, все понятно. Аналогичным способом происходит построение оставшихся трех сторон куба. Код, инициализирующий наши вершины в структуре `CUSTOMVERTEX Vershin`, будет выглядеть следующим образом:

```
CUSTOMVERTEX Vershin[] =
```

```
{
    {1.0f, -1.0f, 1.0f, 0x00000fff}, // A
    {-1.0f, -1.0f, 1.0f, 0x00000fff}, // B
    {-1.0f, 1.0f, 1.0f, 0x00000fff}, // C

    {-1.0f, 1.0f, 1.0f, 0x00000fff}, // C
    {1.0f, 1.0f, 1.0f, 0x00000fff}, // D
    {1.0f, -1.0f, 1.0f, 0x00000fff}, // A

    {-1.0f, -1.0f, 1.0f, 0x00000fff}, // A2
    {-1.0f, -1.0f, -1.0f, 0x00000fff}, // B2
    {-1.0f, 1.0f, -1.0f, 0x00000fff}, // C2
```

```

{-1.0f, 1.0f, -1.0f, 0x00000fff}, // C2
{-1.0f, 1.0f, 1.0f, 0x00000fff}, // D2
{-1.0f, -1.0f, 1.0f, 0x00000fff}, // A2

{1.0f, -1.0f, -1.0f, 0x00000fff}, // A3
{-1.0f, -1.0f, -1.0f, 0x00000fff}, // B3
{-1.0f, 1.0f, -1.0f, 0x00000fff}, // C3

{-1.0f, 1.0f, -1.0f, 0x00000fff}, // C3
{1.0f, 1.0f, -1.0f, 0x00000fff}, // D3
{1.0f, -1.0f, -1.0f, 0x00000fff}, // A3

{1.0f, -1.0f, -1.0f, 0x00000fff}, // A4
{1.0f, -1.0f, 1.0f, 0x00000fff}, // B4
{1.0f, 1.0f, 1.0f, 0x00000fff}, // C4

{1.0f, 1.0f, 1.0f, 0x00000fff}, // C4
{1.0f, 1.0f, -1.0f, 0x00000fff}, // D4
{1.0f, -1.0f, -1.0f, 0x00000fff}, // A4

{1.0f, 1.0f, 1.0f, 0x00000fff}, // A5
{-1.0f, 1.0f, 1.0f, 0x00000fff}, // B5
{-1.0f, 1.0f, -1.0f, 0x00000fff}, // C5

{-1.0f, 1.0f, -1.0f, 0x00000fff}, // C5
{1.0f, 1.0f, -1.0f, 0x00000fff}, // D5
{1.0f, 1.0f, 1.0f, 0x00000fff}, // A5

{1.0f, -1.0f, 1.0f, 0x00000fff}, // A6
{-1.0f, -1.0f, 1.0f, 0x00000fff}, // B6
{-1.0f, -1.0f, -1.0f, 0x00000fff}, // C6

{-1.0f, -1.0f, -1.0f, 0x00000fff}, // C6
{1.0f, -1.0f, -1.0f, 0x00000fff}, // D6
{1.0f, -1.0f, 1.0f, 0x00000fff}, // A6

```

```
};
```

Еще необходимо изменить два значения в функции `InitialBufferVershin()` — там, где с помощью `CreateVertexBuffer()` создается буфер вершин для двух треугольников и, соответственно, шести вершин, мы ставили число 6, умножая его на `sizeof(CUSTOMVERTEX)`. Необходимо изменить число 6 на число 36, т. е. 12 треугольников по 3 вершины, при умножении значений будет 36. А также в функции `DrawPrimitive()` изменится количество треугольников с двух до двенадцати, необходимых для прорисовки куба. В папке `Urok6` на компакт-диске к книге в файле `MatrixCube.cpp` вы найдете весь исходный код.

Индексация вершин

Теперь, как и было обещано, создадим тот же самый куб, но уже с использованием индексного буфера, суть которого заключается в нумерации всех вершин с последовательным их применением. Вы заметили, что когда создается квадрат из двух треугольников, то две вершины имеют одинаковые координаты, дублируя друг друга, а если посмотреть на куб, то их станет в несколько раз больше. Для того чтобы этого не происходило, применяется индексация вершин. При создании квадрата из двух треугольников вы указываете координаты только четырех вершин: A, B, C, D, создающих этот квадрат, и назначаете последовательно номера вершин, сохраняя их в отдельных переменных. Посмотрим на рис. 6.4.

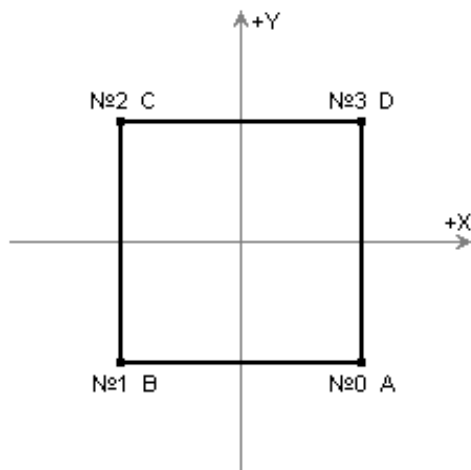


Рис. 6.4. Индексация вершин

Координаты четырех вершин передней стороны куба следующие: $A = (1.0f, -1.0f, -1.0f)$, $B = (-1.0f, -1.0f, -1.0f)$, $C = (-1.0f, 1.0f, -1.0f)$, $D = (1.0f, 1.0f, -1.0f)$.

Индексы для четырех вершин зададим так: 0, 1, 2 — для первого треугольника и 2, 3, 0 — для второго. Опишем переменную, хранящую эти данные:

```
const unsigned short Index[]={
    0,1,2,          2,3,0,
    4,5,6,          6,7,4,
    8,9,10,         10,11,8,
    12,13,14,       14,15,12,
    16,17,18,       18,19,16,
    20,21,22,       22,23,20,
};
```

Далее с помощью средств, имеющихся в Direct3D, объединим эти данные и создадим квадрат. Для каждого квадрата теперь имеется по четыре индекса.

С теорией все, приступим к "кодингу". Сразу замечу — пример, реализующий индексацию вершин, называется Cube.cpp и находится в каталоге Urok6 на компакт-диске. Все изменения коснутся в основном функции InitialBufferVershin(), ее я предлагаю переименовать в InitialObject(), поскольку в ней будет происходить создание как буфера вершин, так и индексного буфера.

В глобальных переменных создадим новую переменную, где будем хранить указатель на интерфейс буфера индексов:

```
LPDIRECT3DINDEXBUFFER9 pBufferIndex = NULL;
```

В переименованной функции InitialObject() изменится структура CUSTOMVERTEX Vershin, содержащая теперь координаты шести квадратов, но уже по четыре вершины на каждый:

```
CUSTOMVERTEX Vershin[] =
{
    {1.0f, -1.0f, -1.0f, 0x00000fff}, // A
    {-1.0f, -1.0f, -1.0f, 0x00000fff}, // B
    {-1.0f, 1.0f, -1.0f, 0x00000fff}, // C
    {1.0f, 1.0f, -1.0f, 0x00000fff}, // D

    {-1.0f, -1.0f, 1.0f, 0x00000fff}, // A2
    {-1.0f, -1.0f, -1.0f, 0x00000fff}, // B2
    {-1.0f, 1.0f, -1.0f, 0x00000fff}, // C2
    {-1.0f, 1.0f, 1.0f, 0x00000fff}, // D2

    {1.0f, -1.0f, 1.0f, 0x00000fff}, // A3
    {-1.0f, -1.0f, 1.0f, 0x00000fff}, // B3
```

```

{-1.0f, 1.0f, 1.0f, 0x00000fff},      // C3
{ 1.0f, 1.0f, 1.0f, 0x00000fff},      // D3

{1.0f, -1.0f, -1.0f, 0x00000fff},      // A4
{1.0f, -1.0f, 1.0f, 0x00000fff},      // B4
{1.0f, 1.0f, 1.0f, 0x00000fff},      // C4
{1.0f, 1.0f, -1.0f, 0x00000fff},      // D4

{1.0f, 1.0f, 1.0f, 0x00000fff},      // A5
{-1.0f, 1.0f, 1.0f, 0x00000fff},      // B5
{-1.0f, 1.0f, -1.0f, 0x00000fff},      // C5
{1.0f, 1.0f, -1.0f, 0x00000fff},      // D5

{1.0f, -1.0f, 1.0f, 0x00000fff},      // A6
{-1.0f, -1.0f, 1.0f, 0x00000fff},      // B6
{-1.0f, -1.0f, -1.0f, 0x00000fff},      // C6
{1.0f, -1.0f, -1.0f, 0x00000fff},      // D6
};

```

Все дальнейшие действия сводятся, как и в случае с буфером вершин, к созданию индексного буфера, его блокировке, записи данных и разблокировке. Соответственно то же самое будет происходить и с буфером вершин. Для этих целей имеются следующие функции:

- ❑ IDirect3DDevice9::CreateIndexBuffer;
- ❑ IDirect3DIndexBuffer9::Lock;
- ❑ IDirect3DIndexBuffer9::Unlock.

Обратите внимание, что при блокировке и разблокировке используются функции, определенные в интерфейсе IDirect3DIndexBuffer9, но по своим параметрам они ничем не отличаются от одноименных функций для интерфейса IDirect3DVertexBuffer9. Создадим индексный буфер, а потом рассмотрим все его составляющие:

```

pDirect3DDevice -> CreateIndexBuffer(36 * sizeof(Index), 0,
                                     D3DFMT_INDEX16, D3DPOOL_DEFAULT, &pBufferIndex, NULL);

```

Прототип функции CreateIndexBuffer() выглядит так:

```

HRESULT CreateIndexBuffer(
    UINT           Length,
    DWORD          Usage,
    D3DFORMAT      Format,

```

```

D3DPOOL          Pool,
IDirect3DIndexBuffer** ppIndexBuffer,
HANDLE*          pHandle);

```

Функция `CreateIndexBuffer()` имеет следующие параметры:

- `Length` — размер индексного буфера в байтах;
- `Usage` — способ применения создаваемого буфера индексов;
- `Format` — член `D3DFORMAT`, описывает формат индексного буфера. Для его определения применяются два флага: `D3DFM_INDEX16` — каждый индекс занимает 16 битов, `D3DFM_INDEX32` — каждый индекс занимает 32 бита;
- `Pool` — член `D3DPOOL`, описывает формат для размещения в памяти;
- `ppIndexBuffer` — адрес указателя, в котором будет храниться создаваемый индексный буфер;
- `pHandle` — зарезервированный параметр, устанавливается в `NULL`.

Эта функция практически идентична функции по созданию буфера вершин, поэтому нет смысла останавливаться на ней.

Далее созданный буфер необходимо заблокировать для заполнения данными и разблокировать:

```

VOID* pBI;
pBufferIndex -> Lock(0, sizeof(Index), (void**)&pBI, 0);
memcpy(pBI, Index, sizeof(Index));
pBufferIndex -> Unlock();

```

Все действия, ранее производимые с буфером вершин, не изменяются.

И последнее нововведение произойдет в функции `RenderingDirect3D()`, в месте, где происходит прорисовка сцены, добавится строка для установки индексного буфера с помощью функции `IDirect3DDevice9::SetIndices`:

```
pDirect3DDevice -> SetIndices(pBufferIndex);
```

Прототип функции `SetIndices()` выглядит таким образом:

```
HRESULT SetIndices(IDirect3DIndexBuffer9* pIndexData);
```

Единственный параметр `pIndexData` служит для обозначения буфера индексов.

А взамен функции `DrawPrimitive()`, необходимой для вывода примитива на экран, появится `IDirect3DDevice9::DrawIndexPrimitive`, работа которой сводится к выводу примитива, но на основе проиндексированных вершин. Рассмотрим ее прототип:

```

HRESULT DrawIndexPrimitive(
    D3DPRIMITIVETYPE Type,
    INT                BaseVertexIndex,

```

```

UINT          MinIndex,
UINT          NumVertices,
UINT          StartIndex,
UINT          PrimitiveCount);

```

Функция имеет следующие параметры:

- ❑ **Type** — член `D3DPRIMITIVETYPE`, описывает необходимый тип примитива. У нас по-прежнему `D3DPT_TRIANGLELIST`;
- ❑ **BaseVertexIndex** — смещение индекса. Установим нулевое значение;
- ❑ **MinIndex** — минимальный индекс. Установим нулевое значение;
- ❑ **NumVertices** — количество рисуемых вершин, у нас 36;
- ❑ **StartIndex** — стартовый индекс, с которого начинается построение вершин, установим нулевое значение;
- ❑ **PrimitiveCount** — количество рисуемых треугольников, у нас 12.

Вызов самой функции будет иметь вид:

```
pDirect3DDevice->DrawIndexPrimitive(D3DPT_TRIANGLELIST, 0, 0, 36, 0, 12);
```

На этом весь процесс по построению куба с помощью проиндексированных вершин заканчивается. Единственное, что еще необходимо сделать, — это освободить захваченные ресурсы для буфера индексов:

```

if (pBufferIndex != NULL)
pBufferIndex -> Release();

```

Да, и главное — если вы изменили название функции `InitialBufferVershin()` на `InitialObject()` (как я предлагал), не забудьте внести изменения также в `WinMain()`.

Все, что получилось, приведено в листинге 6.2.

Листинг 6.2. Файл `Cube.cpp`

```

//-----
//  Cube.cpp
//  создаем куб с помощью буфера индексов
//-----
#include <windows.h>    // подключаем заголовочный файл Windows
#include <d3d9.h>        // подключаем заголовочный файл DirectX 9 SDK
#include <d3dx9.h>       // подключаем заголовочный файл из D3DX-утилит
#include <mmsystem.h>    // подключаем системную библиотеку
//-----
//  глобальные переменные

```

```
//-----
LPDIRECT3D9 pDirect3D          = NULL;    // объект в Direct3D
LPDIRECT3DDEVICE9 pDirect3DDevice = NULL;  // устройство
LPDIRECT3DVERTEXBUFFER9 pBufferVershin = NULL; // буфер вершин
LPDIRECT3DINDEXBUFFER9 pBufferIndex   = NULL; // индексный буфер
struct CUSTOMVERTEX
{
    FLOAT X, Y, Z;    // координаты вершин
    DWORD color;      // цвет вершин
};
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE) // формат вершин
//-----
// функция
// IDirect3D()
// инициализация Direct3D
//-----
HRESULT InitialDirect3D(HWND hwnd)
{
    if (NULL == (pDirect3D = Direct3DCreate9(D3D_SDK_VERSION)))
        return E_FAIL;
    D3DDISPLAYMODE Display;
    if (FAILED(pDirect3D -> GetAdapterDisplayMode(
        D3DADAPTER_DEFAULT, &Display)))
        return E_FAIL;
    D3DPRESENT_PARAMETERS Direct3DParametr;
    ZeroMemory(&Direct3DParametr, sizeof(Direct3DParametr));
    Direct3DParametr.Windowed = TRUE;
    Direct3DParametr.SwapEffect = D3DSWAPEFFECT_DISCARD;
    Direct3DParametr.BackBufferFormat = Display.Format;
    if (FAILED(pDirect3D -> CreateDevice(D3DADAPTER_DEFAULT,
        D3DDEVTYPE_HAL, hwnd, D3DCREATE_SOFTWARE_VERTEXPROCESSING,
        &Direct3DParametr, &pDirect3DDevice)))
        return E_FAIL;
    pDirect3DDevice -> SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE);
    // отключаем освещение Direct3D
    pDirect3DDevice -> SetRenderState(D3DRS_LIGHTING, FALSE);
    return S_OK;
}
```

```
//-----  
//  функция  
//  InitialObject()  
//  создание объекта  
//-----  
HRESULT InitialObject()  
{  
    CUSTOMVERTEX Vershin[] =  
    {  
        {1.0f, -1.0f, -1.0f, 0x00000fff},    //  A  
        {-1.0f, -1.0f, -1.0f, 0x00000fff},    //  B  
        {-1.0f, 1.0f, -1.0f, 0x00000fff},    //  C  
        {1.0f, 1.0f, -1.0f, 0x00000fff},    //  D  
  
        {-1.0f, -1.0f, 1.0f, 0x00000fff},    //  A2  
        {-1.0f, -1.0f, -1.0f, 0x00000fff},    //  B2  
        {-1.0f, 1.0f, -1.0f, 0x00000fff},    //  C2  
        {-1.0f, 1.0f, 1.0f, 0x00000fff},    //  D2  
  
        {1.0f, -1.0f, 1.0f, 0x00000fff},    //  A3  
        {-1.0f, -1.0f, 1.0f, 0x00000fff},    //  B3  
        {-1.0f, 1.0f, 1.0f, 0x00000fff},    //  C3  
        {1.0f, 1.0f, 1.0f, 0x00000fff},    //  D3  
  
        {1.0f, -1.0f, -1.0f, 0x00000fff},    //  A4  
        {1.0f, -1.0f, 1.0f, 0x00000fff},    //  B4  
        {1.0f, 1.0f, 1.0f, 0x00000fff},    //  C4  
        {1.0f, 1.0f, -1.0f, 0x00000fff},    //  D4  
  
        {1.0f, 1.0f, 1.0f, 0x00000fff},    //  A5  
        {-1.0f, 1.0f, 1.0f, 0x00000fff},    //  B5  
        {-1.0f, 1.0f, -1.0f, 0x00000fff},    //  C5  
        {1.0f, 1.0f, -1.0f, 0x00000fff},    //  D5  
  
        {1.0f, -1.0f, 1.0f, 0x00000fff},    //  A6  
        {-1.0f, -1.0f, 1.0f, 0x00000fff},    //  B6  
        {-1.0f, -1.0f, -1.0f, 0x00000fff},    //  C6  
        {1.0f, -1.0f, -1.0f, 0x00000fff},    //  D6
```

```

};
const unsigned short Index[]={
0,1,2,      2,3,0,
4,5,6,      6,7,4,
8,9,10,     10,11,8,
12,13,14,   14,15,12,
16,17,18,   18,19,16,
20,21,22,   22,23,20,
};
// создаем буфер вершин
if (FAILED (pDirect3DDevice->
            CreateVertexBuffer (36*sizeof(CUSTOMVERTEX),
                                0, D3DFVF_CUSTOMVERTEX, D3DPOOL_DEFAULT, &pBufferVershin, NULL)))
return E_FAIL;
// блокируем
VOID* pBV;
if (FAILED(pBufferVershin -> Lock 0,sizeof(Vershin),
            (void*)&pBV, 0)))

return E_FAIL;
// копируем
memcpy(pBV, Vershin, sizeof(Vershin));
// разблокируем
pBufferVershin -> Unlock();
// создаем индексный буфер
pDirect3DDevice -> CreateIndexBuffer(36 * sizeof(Index), 0,
                                     D3DFMT_INDEX16, D3DPOOL_DEFAULT, &pBufferIndex, NULL);
// блокируем
VOID* pBI;
pBufferIndex -> Lock(0, sizeof(Index), (void*)&pBI, 0);
// копируем
memcpy(pBI, Index, sizeof(Index));
// разблокируем
pBufferIndex -> Unlock();
return S_OK;
}
//-----
// функция
// Matrix()
// мировая матрица, матрица вида, матрица проекции

```

```

//-----
VOID Matrix()
{
    D3DXMATRIX MatrixWorld;           // мировая матрица
    D3DXMATRIX MatrixView;            // матрица вида
    D3DXMATRIX MatrixProjection;      // матрица проекции
    // MatrixWorld
    UINT Time = timeGetTime() % 5000;
    FLOAT Angle = Time * (2.0f * D3DX_PI) / 5000.0f;
    D3DXMatrixRotationX(&MatrixWorld, Angle);
    pDirect3DDevice -> SetTransform(D3DTS_WORLD, &MatrixWorld);
    // MatrixView
    D3DXMatrixLookAtLH(&MatrixView, &D3DXVECTOR3(0.0f, 0.0f, -8.0f),
                      &D3DXVECTOR3(0.0f, 0.0f, 0.0f),
                      &D3DXVECTOR3(0.0f, 1.0f, 0.0f));
    pDirect3DDevice -> SetTransform(D3DTS_VIEW, &MatrixView);
    // MatrixProjection
    D3DXMatrixPerspectiveFovLH(&MatrixProjection, D3DX_PI/4, 1,1, 100.0f);
    pDirect3DDevice -> SetTransform(D3DTS_PROJECTION, &MatrixProjection);
}
//-----
// функция
// RenderingDirect3D()
// рисование
//-----
VOID RenderingDirect3D()
{
    if (pDirect3DDevice == NULL)      // проверяем ошибки
        return;
    pDirect3DDevice -> Clear(0, NULL, D3DCLEAR_TARGET,
                           D3DCOLOR_XRGB(255, 255, 0), 1.0f, 0);

    // начало сцены
    pDirect3DDevice -> BeginScene();
    // здесь происходит прорисовка сцены
    Matrix();
    pDirect3DDevice -> SetStreamSource(0, pBufferVershin, 0,
                                       sizeof(CUSTOMVERTEX));
    pDirect3DDevice -> SetFVF(D3DFVF_CUSTOMVERTEX);
    pDirect3DDevice -> SetIndices(pBufferIndex);
}

```

```

// вывод объекта
pDirect3DDevice -> DrawIndexedPrimitive(D3DPT_TRIANGLELIST,
                                         0, 0, 360, 12);

// конец сцены
pDirect3DDevice -> EndScene();
pDirect3DDevice -> Present(NULL, NULL, NULL, NULL);
}

//-----
// функция
// DeleteDirect3D()
// освобождение захваченных ресурсов
//-----
VOID DeleteDirect3D()
{
    if (pBufferIndex != NULL)
        pBufferIndex -> Release();

    if (pBufferVershin != NULL)
        pBufferVershin -> Release();

    if (pDirect3DDevice != NULL)
        pDirect3DDevice -> Release();

    if (pDirect3D != NULL)
        pDirect3D -> Release();
}

//-----
// функция
// MainWinProc()
// здесь происходит обработка сообщений
//-----
LRESULT CALLBACK MainWinProc(HWND hwnd,
                              UINT msg,
                              WPARAM wparam,
                              LPARAM lparam)
{
    switch(msg)
    {

```

```

        case WM_DESTROY:
        {
            DeleteDirect3D();
            PostQuitMessage(0);
            return(0);
        }
    }
    return DefWindowProc(hwnd, msg, wparam, lparam);
}

//-----
//  функция
//  WinMain
//  входная точка приложения
//-----

int WINAPI WinMain(HINSTANCE hinstance,
                  HINSTANCE hprevinstance,
                  LPSTR lpcmdline,
                  int ncmdshow)
{
    WNDCLASSEX windowclass;    // создаем класс
    HWND hwnd;                // создаем дескриптор окна
    MSG msg;                   // создаем идентификатор сообщения
    // определим класс окна WNDCLASSEX
    windowclass.cbSize = sizeof(WNDCLASSEX);
    windowclass.style = CS_DBLCLKS|CS_OWNDC|CS_HREDRAW|CS_VREDRAW;
    windowclass.lpfnWndProc = MainWinProc;
    windowclass.cbClsExtra = 0;
    windowclass.cbWndExtra = 0;
    windowclass.hInstance = hinstance;
    windowclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    windowclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    windowclass.hbrBackground = (HBRUSH)GetStockObject(GRAY_BRUSH);
    windowclass.lpszMenuName = NULL;
    windowclass.lpszClassName = "WINDOWSCCLASS";
    windowclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
    // регистрируем класс
    if (!RegisterClassEx(&windowclass))
        return(0);
}

```

```
// теперь, когда класс зарегистрирован, можно создать окно
if (!hwnd = CreateWindowEx(NULL, "WINDOWSCCLASS", "Текст",
                           WS_OVERLAPPEDWINDOW|WS_VISIBLE, 300, 150,
                           500, 400, NULL, NULL, hinstance, NULL))

return(0);
if (SUCCEEDED(InitialDirect3D(hwnd)))
{
    if (SUCCEEDED(InitialObject() ))
    {
        ShowWindow(hwnd, SW_SHOWDEFAULT);
        UpdateWindow(hwnd);
        ZeroMemory(&msg, sizeof(msg));
        while(msg.message != WM_QUIT)
        {
            if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
            {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
            }
            else
                RenderingDirect3D();
        }
    }
}
return 0;
}
```

Итоги урока

На основании матричных преобразований и индексации вершин на этом уроке был создан куб. Все стороны окрашены в синий цвет и задан угол вращения. На *уроке 7* мы сделаем куб более реалистичным, а также раскрасим стороны куба в разные цвета.

Урок 7



Буфер глубины или Z-буфер

На *уроке 6* мы создали куб и вывели его на экран. Но вот сам объект оставил двоякое впечатление, не правда ли? Действительно, рисуемый куб вроде бы и был кубом, но выглядел совсем не 3D-объектом. На самом деле это был все тот же 2D-примитив, но построенный и отображенный как 3D-объект. Все дело в том, что мы не применяли мощную систему освещения, предлагаемую Direct3D, и, что самое главное, не был использован буфер глубины, называемый еще Z-буфером. Когда на предыдущем уроке происходила прорисовка куба, то все стороны накладывались друг на друга в определенном нами порядке. Для того чтобы действительно получился трехмерный объект, необходимо применить Z-буфер, с помощью которого происходит правильная расстановка всех вершин. Задняя сторона куба будет удалена от наших глаз и покажется меньше, как в реальном мире. То есть само построение примет более реалистичный вид и приобретет объемность. Все это достигается за счет Z-буферизации, с ее помощью происходит расстановка вершин в пространстве относительно камеры. Освещение и материал будут обсуждаться на *уроке 8*, поэтому мы немного схитрим и раскрасим каждую сторону куба разным цветом с помощью изменения значения параметра `color` в структуре `CUSTOMVERTEX`. Но сначала обратимся к функции `InitialDirect3D()`, где появятся новые параметры, необходимые для отображения Z-буфера.

На *уроке 3* был создан объект `Direct3DParametr` структуры `D3DPRESENT_PARAMETERS`. К трем параметрам представления добавятся два новых параметра, создающие Z-буфер:

```
Direct3DParametr.EnableAutoDepthStencil = TRUE;  
Direct3DParametr.AutoDepthStencilFormat = D3DFMT_D16;
```

В первой строке параметр `EnableAutoDepthStencil` принимает значение `TRUE`, показывающее возможность создания и установки Z-буфера для устройства Direct3D. Сам Z-буфер создается автоматически каждый раз при изменении размера окна.

Во второй строке устанавливается формат поверхности Z-буфера. Это значение — член структуры `D3DFORMAT`, которую мы разбирали на *уроке 3*. Обычно формат определяется флагом `D3DFMT_D16`.

Таким образом задается формат буфера глубины. Логическим завершением всей функции будет новая строка для создания буфера глубины:

```
pDirect3DDevice -> SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
```

Первый параметр в функции `SetRenderState()` является членом перечисляемого типа `D3DRENDERSTATETYPE`, определяющего тип отображения для устройства `Direct3D`. До этого момента уже применялись значения из этого типа, но подробное его рассмотрение тогда не имело смысла, а сейчас самое время. Сам тип очень большой, его прототип приведу полностью, а разбирать будем только необходимые для нас параметры:

```
typedef enum _D3DRENDERSTATETYPE {
    D3DRS_ZENABLE                = 7,
    D3DRS_FILLMODE               = 8,
    D3DRS_SHADEMODE              = 9,
    D3DRS_LINEPATTERN            = 10,
    D3DRS_ZWRITEENABLE           = 14,
    D3DRS_ALPHATESTENABLE        = 15,
    D3DRS_LASTPIXEL              = 16,
    D3DRS_SRCBLEND                = 19,
    D3DRS_DESTBLEND              = 20,
    D3DRS_CULLMODE               = 22,
    D3DRS_ZFUNC                  = 23,
    D3DRS_ALPHAREF               = 24,
    D3DRS_ALPHAFUNC              = 25,
    D3DRS_DITHERENABLE          = 26,
    D3DRS_ALPHABLENDENABLE       = 27,
    D3DRS_FOGENABLE              = 28,
    D3DRS_SPECULARENABLE        = 29,
    D3DRS_ZVISIBLE               = 30,
    D3DRS_FOGCOLOR               = 34,
    D3DRS_FOGTABLEMODE           = 35,
    D3DRS_FOGSTART               = 36,
    D3DRS_FOGEND                 = 37,
    D3DRS_FOGDENSITY             = 38,
    D3DRS_EDGEANTIALIAS          = 40,
    D3DRS_ZBIAS                  = 47,
    D3DRS_RANGEFOGENABLE         = 48,
```

D3DRS_STENCILENABLE	= 52,
D3DRS_STENCILFAIL	= 53,
D3DRS_STENCILZFAIL	= 54,
D3DRS_STENCILPASS	= 55,
D3DRS_STENCILFUNC	= 56,
D3DRS_STENCILREF	= 57,
D3DRS_STENCILMASK	= 58,
D3DRS_STENCILWRITEMASK	= 59,
D3DRS_TEXTUREFACTOR	= 60,
D3DRS_WRAP0	= 128,
D3DRS_WRAP1	= 129,
D3DRS_WRAP2	= 130,
D3DRS_WRAP3	= 131,
D3DRS_WRAP4	= 132,
D3DRS_WRAP5	= 133,
D3DRS_WRAP6	= 134,
D3DRS_WRAP7	= 135,
D3DRS_CLIPPING	= 136,
D3DRS_LIGHTING	= 137,
D3DRS_AMBIENT	= 139,
D3DRS_FOGVERTEXMODE	= 140,
D3DRS_COLORVERTEX	= 141,
D3DRS_LOCALVIEWER	= 142,
D3DRS_NORMALIZENORMALS	= 143,
D3DRS_DIFFUSEMATERIALSOURCE	= 145,
D3DRS_SPECULARMATERIALSOURCE	= 146,
D3DRS_AMBIENTMATERIALSOURCE	= 147,
D3DRS_EMISSIVEMATERIALSOURCE	= 148,
D3DRS_VERTEXBLEND	= 151,
D3DRS_CLIPPLANEENABLE	= 152,
D3DRS_SOFTWAREVERTEXPROCESSING	= 153,
D3DRS_POINTSIZE	= 154,
D3DRS_POINTSIZE_MIN	= 155,
D3DRS_POINTSPRITEENABLE	= 156,
D3DRS_POINTSCALEENABLE	= 157,
D3DRS_POINTSCALE_A	= 158,
D3DRS_POINTSCALE_B	= 159,
D3DRS_POINTSCALE_C	= 160,
D3DRS_MULTISAMPLEANTIALIAS	= 161,

```

D3DRS_MULTISAMPLEMASK          = 162,
D3DRS_PATCHEDGESTYLE          = 163,
D3DRS_PATCHSEGMENTS           = 164,
D3DRS_DEBUGMONITORTOKEN       = 165,
D3DRS_POINTSIZE_MAX           = 166,
D3DRS_INDEXEDVERTEXBLENDENABLE = 167,
D3DRS_COLORWRITEENABLE        = 168,
D3DRS_TWEENFACTOR              = 170,
D3DRS_BLENDOP                  = 171,
D3DRS_POSITIONORDER           = 172,
D3DRS_NORMALORDER              = 173,
D3DRS_FORCE_DWORD              = 0x7fffffff
} D3DRENDERSTATETYPE;

```

Рассмотрим некоторые параметры типа `D3DRENDERSTATETYPE`:

- ❑ `D3DRS_ZENABLE` — тип Z-буфера. Во втором параметре функции `SetRenderState()` определяется способ работы первого параметра. Значение `D3DRS_ZENABLE` включает Z-буфер;
- ❑ `D3DRS_CULLMODE` — это способ отображения задних сторон примитива, определяется вторым параметром функции;
- ❑ `D3DRS_LIGHTING` — освещение сцены. При помощи значений `TRUE` и `FALSE` во втором параметре функции соответственно включают и отключают свет. На этом уроке освещение отключено, поэтому `FALSE`;
- ❑ `D3DRS_AMBIENT` — окружающее освещение (подсветка). Значение по умолчанию 0, определяется во втором параметре функции `SetRenderState()`.

Во второй строке функции `SetRenderState()` определяется режим работы первого параметра. Применяемое значение `D3DZB_TRUE` принадлежит перечисляемому типу `D3DZBUFFERTYPE`, используемому для определения константы формата глубины. Рассмотрим запись типа `D3DZBUFFERTYPE`:

```

typedef enum _D3DZBUFFERTYPE {
    D3DZB_FALSE      = 0;
    D3DZB_TRUE        = 1;
    D3DZB_USW         = 2;
    D3DZB_FORCE_DWORD = 0x7fffffff
} D3DZBUFFERTYPE;

```

Тип `D3DZBUFFERTYPE` имеет следующие параметры:

- ❑ `D3DZB_FALSE` — не разрешает использовать Z-буфер;
- ❑ `D3DZB_TRUE` — разрешает использовать Z-буфер;

- D3DZB_USW — разрешает использовать W-буфер;
- D3DZB_FORSE_DWORD — значение не используется.

Итак, мы создали и включили буфер глубины, последним шагом, определяющим его работу, будет добавление флага D3DCLEAR_ZBUFFER в функции Clear(), внутри созданной нами ранее RenderingDirect3D():

```
pDirect3DDevice -> Clear(0, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,  
                        D3DCOLOR_XRGB(60, 100, 150), 1.0f, 0);
```

С помощью добавленного флага D3DCLEAR_ZBUFFER мы очищаем заодно и буфер глубины посредством перечисления, используя побитовое ИЛИ (значок "|"). На предыдущих уроках мы очищали или закрашивали задний буфер в ярко-желтый цвет, сейчас предлагаемый фон специально был сделан темнее с помощью значения D3DCOLOR_XRGB (60, 100, 150). Когда мы впервые только создавали это приложение, освещение всей сцены отключалось, и чтобы хоть как-то мрачную картину сделать светлее, предлагался ярко-желтый фон. Здесь же мы раскрасим куб, а точнее стороны куба в разные цвета, поэтому фон сделаем более темным.

Давайте вернемся в функцию InitialDirect3D() и разберем следующую строку:

```
pDirect3DDevice -> SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE);
```

С помощью этой операции происходит отключение отсеечения невидимых сторон куба, т. е. задних. Сейчас пришло время включить отсеечение, в Direct3D это нормальная практика. Невидимые задние стороны не рисуются, а отсекаются, экономя при этом системные ресурсы и время на прорисовку. Тем более что их все равно не видно. Для того чтобы включить режим отсеечения, необходимо задействовать второй параметр в функции SetRenderState(), являющийся перечисляемым типом D3DCULL. Вот как описывается его прототип:

```
typedef enum _D3DCULL {  
    D3DCULL_NONE          = 1;  
    D3DCULL_CW            = 2;  
    D3DCULL_CCW           = 3;  
    D3DCULL_FORCE_DWORD = 0x7fffffff  
} D3DCULL;
```

Тип D3DCULL имеет следующие параметры:

- D3DCULL_NONE — отсеечение задних граней отключено;
- D3DCULL_CW — отсеечение включено и происходит по часовой стрелке;
- D3DCULL_CCW — отсеечение включено и происходит против часовой стрелки;
- D3DCULL_FORCE_DWORD — значение не используется.

При определении любого из трех значений в функции `SetRenderState()` задается режим отсечения. Поскольку в нашем примере построение треугольников происходит по часовой стрелке, выбираем `D3DCULL_CW`.

Связываем стороны куба

Как только вы осуществите все установки, рассмотренные в предыдущем разделе, и откомпилируете приложение, результат вас неприятно удивит. Все, что вы увидите на экране, — это непонятно как отсекающиеся стороны куба, вращающиеся на экране, а созданный куб будет просто изобиловать геометрическими дефектами! Давайте разберемся, в чем дело и что именно было сделано неправильно. На самом деле ошибка возникает при инициализации вершин куба. Весь цикл по отсечению может происходить по часовой стрелке (по умолчанию) и наоборот, для связанных, повторяю, связанных треугольников или сторон куба. Не забывайте, что каждая сторона все равно состоит из двух треугольников. Когда на *уроке 6* создавался куб с помощью функции `InitialObject()` и структуры `Vershin`, все вроде бы делалось правильно, но вот о связанности треугольников и сторон куба речи не шло. Чтобы понять, как это достигается, давайте посмотрим на рис. 7.1.

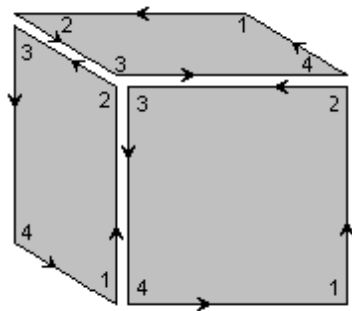


Рис. 7.1. Связанные стороны куба

Стартуем мы в точке 1 лицевой стороны куба, финишируем в точке 4. Вектор от точки 4 до 1 дорисовывается автоматически, но мы все равно остаемся в точке 4, которая, в свою очередь, будет являться начальной точкой для левой стороны куба (она соответствует точке 1). Аналогично достраивается сторона куба и все последующие, строго по часовой стрелке, поскольку при включении отсечения был использован флаг `D3DCULL_CW`. То же самое можно сделать и против часовой стрелки, но в этом случае необходимо применить флаг `D3DCULL_CCW`. Таким образом, создается связанность всех сторон куба в единое целое.

В связи с тем, что у нас теперь изменились координаты для вершин, посмотрим на структуру `Vershin`, где также меняется цвет каждой стороны куба за счет изменения значения `color`. Здесь вы можете раскрасить стороны куба как угодно, но мне захотелось именно так:

```
HRESULT InitialObject()
{
    CUSTOMVERTEX Vershin[] =
    {
        // синий
        {1.0f, -1.0f, -1.0f, 0x00000fff}, // A
        {1.0f, 1.0f, -1.0f, 0x00000fff}, // B
        {-1.0f, 1.0f, -1.0f, 0x00000fff}, // C
        {-1.0f, -1.0f, -1.0f, 0x00000fff}, // D
        // красный
        {-1.0f, -1.0f, 1.0f, 0xffff0000}, // A2
        {-1.0f, 1.0f, 1.0f, 0xffff0000}, // B2
        {-1.0f, 1.0f, -1.0f, 0xffff0000}, // C2
        {-1.0f, -1.0f, 1.0f, 0xffff0000}, // D2
        // синий
        {-1.0f, -1.0f, 1.0f, 0x00000fff}, // A3
        {-1.0f, 1.0f, 1.0f, 0x00000fff}, // B3
        {1.0f, 1.0f, 1.0f, 0x00000fff}, // C3
        {1.0f, -1.0f, 1.0f, 0x00000fff}, // D3
        // зеленый
        {1.0f, -1.0f, 1.0f, 0xff0fff00}, // A4
        {1.0f, 1.0f, 1.0f, 0xff0fff00}, // B4
        {1.0f, 1.0f, -1.0f, 0xff0fff00}, // C4
        {1.0f, -1.0f, -1.0f, 0xff0fff00}, // D4
        // розовый
        {1.0f, -1.0f, -1.0f, 0xffff000ff}, // A5
        {-1.0f, -1.0f, -1.0f, 0xffff000ff}, // B5
        {-1.0f, -1.0f, 1.0f, 0xffff000ff}, // C5
        {1.0f, -1.0f, 1.0f, 0xffff000ff}, // D5
        // розовый
        {1.0f, 1.0f, 1.0f, 0xffff000ff}, // A6
        {-1.0f, 1.0f, 1.0f, 0xffff000ff}, // B6
        {-1.0f, 1.0f, -1.0f, 0xffff000ff}, // C6
        {1.0f, 1.0f, -1.0f, 0xffff000ff}, // D6
    };
}
```

```
const unsigned short Index[]={
    0,1,2,      2,3,0,
    4,5,6,      6,7,4,
    8,9,10,     10,11,8,
    12,13,14,   14,15,12,
    16,17,18,   18,19,16,
    20,21,22,   22,23,20,
};
```

Индексные значения для буфера индексов остаются такими же. Произведите компиляцию и посмотрите, что получится. Но перед этим я предлагаю внести еще одно изменение, которое коснется определенной нами функции `Matrix()`. На уроке 6 вращение куба было задано по оси *X*, также можно было вращать его по оси *Y* или *Z*. Но можно объединить эти операции, чтобы рисуемый объект вращался в разных направлениях и выглядел более артистично. Этим мы сейчас и займемся.

В функции `Matrix()` добавим две новые мировые матрицы для каждой из осей вращения:

```
D3DXMATRIX MatrixWorld, MatrixWorldX, MatrixWorldY;
```

Потом с помощью двух функций сформируем матрицы для осей *X* и *Y*:

```
D3DXMatrixRotationX(&MatrixWorldX, Angle);
```

```
D3DXMatrixRotationY(&MatrixWorldY, Angle);
```

После чего все результирующие матрицы необходимо перемножить между собой, чтобы получить итоговый результат. Делается это с помощью функции `D3DXMatrixMultiply()`. Рассмотрим ее прототип:

```
D3DXMATRIX *D3DXMatrixMultiply(
    D3DXMATRIX*      pOut,
    CONST D3DXMATRIX* pM1,
    CONST D3DXMATRIX* pM2)
```

Функция `D3DXMatrixMultiply()` имеет следующие параметры:

- ❑ `pOut` — указатель на структуру `D3DXMATRIX`, который является результатом действий по умножению между собой двух матриц;
- ❑ `pM1` — указатель на структуру `D3DXMATRIX`, это матрица `MatrixWorldX`;
- ❑ `pM2` — указатель на структуру `D3DXMATRIX`. Это вторая матрица, участвующая в операции — `MatrixWorldY`.

Получаем такую строку кода:

```
D3DXMatrixMultiply(&MatrixWorld, &MatrixWorldX, &MatrixWorldY);
```

Листинг 7.1. Файл Cube.cpp

[illegible]

```

return E_FAIL;
D3DPRESENT_PARAMETERS Direct3DParametr;
ZeroMemory(&Direct3DParametr, sizeof(Direct3DParametr));
Direct3DParametr.Windowed                = TRUE;
Direct3DParametr.SwapEffect               = D3DSWAPEFFECT_DISCARD;
Direct3DParametr.BackBufferFormat        = Display.Format;
Direct3DParametr.EnableAutoDepthStencil = TRUE;
Direct3DParametr.AutoDepthStencilFormat = D3DFMT_D16;
if (FAILED (pDirect3D -> CreateDevice(D3DADAPTER_DEFAULT,
                                      D3DDEVTYPE_HAL, hwnd,
                                      D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                      &Direct3DParametr, &pDirect3DDevice)))

return E_FAIL;
// включаем отсечение Direct3D
pDirect3DDevice -> SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);
// отключаем освещение Direct3D
pDirect3DDevice -> SetRenderState(D3DRS_LIGHTING, FALSE);
// Включаем Z-буфер
pDirect3DDevice -> SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
return S_OK;
}
//-----
// функция
// InitialObject()
// инициализация вершины
//-----
HRESULT InitialObject()
{
    CUSTOMVERTEX Vershin[] =
    {
        // синий
        {1.0f, -1.0f, -1.0f, 0x00000fff}, // A
        {1.0f, 1.0f, -1.0f, 0x00000fff}, // B
        {-1.0f, 1.0f, -1.0f, 0x00000fff}, // C
        {-1.0f, -1.0f, -1.0f, 0x00000fff}, // D
        // красный
        {-1.0f, -1.0f, 1.0f, 0xffff0000}, // A2
        {-1.0f, 1.0f, 1.0f, 0xffff0000}, // B2
        {1.0f, 1.0f, 1.0f, 0xffff0000}, // C2
        {1.0f, -1.0f, 1.0f, 0xffff0000}, // D2
    }
}

```

```

    // синий
    {-1.0f, -1.0f, 1.0f, 0x00000fff},      // A3
    {-1.0f, 1.0f, 1.0f, 0x00000fff},      // B3
    {1.0f, 1.0f, 1.0f, 0x00000fff},       // C3
    {1.0f, -1.0f, 1.0f, 0x00000fff},      // D3

    // зеленый
    {1.0f, -1.0f, 1.0f, 0xff0fff00},      // A4
    {1.0f, 1.0f, 1.0f, 0xff0fff00},      // B4
    {1.0f, 1.0f, -1.0f, 0xff0fff00},      // C4
    {1.0f, -1.0f, -1.0f, 0xff0fff00},     // D4

    // розовый
    {1.0f, -1.0f, -1.0f, 0xffff000ff},    // A5
    {-1.0f, -1.0f, -1.0f, 0xffff000ff},   // B5
    {-1.0f, -1.0f, 1.0f, 0xffff000ff},    // C5
    {1.0f, -1.0f, 1.0f, 0xffff000ff},     // D5

    // розовый
    {1.0f, 1.0f, 1.0f, 0xffff000ff},      // A6
    {-1.0f, 1.0f, 1.0f, 0xffff000ff},     // B6
    {-1.0f, 1.0f, -1.0f, 0xffff000ff},    // C6
    {1.0f, 1.0f, -1.0f, 0xffff000ff},     // D6
};

const unsigned short Index[]={
    0,1,2,      2,3,0,
    4,5,6,      6,7,4,
    8,9,10,     10,11,8,
    12,13,14,   14,15,12,
    16,17,18,   18,19,16,
    20,21,22,   22,23,20,
};

// создаем буфер вершин
if (FAILED(pDirect3DDevice -> CreateVertexBuffer(
    36 *sizeof(CUSTOMVERTEX), 0,
    D3DFVF_CUSTOMVERTEX,D3DPool_DEFAULT,
    &pBufferVershin, NULL)))

return E_FAIL;

// блокируем
VOID* pBV;
if (FAILED(pBufferVershin -> Lock(0, sizeof(Vershin),
    void**)&pBV, 0)))

```

```

return E_FAIL;
// копируем
memcpy(pBV, Vershin, sizeof(Vershin));
// разблокируем
pBufferVershin -> Unlock();
// создаем индексный буфер
pDirect3DDevice -> CreateIndexBuffer(36 * sizeof(Index),
                                     0, D3DFMT_INDEX16, D3DPOOL_DEFAULT,
                                     &pBufferIndex, NULL);

// блокируем
VOID* pBI;
PBufferIndex -> Lock(0, sizeof(Index), (void*)&pBI, 0);
// копируем
memcpy(pBI, Index, sizeof(Index));
// разблокируем
pBufferIndex -> Unlock();
return S_OK;
}

//-----
// функция
// Matrix()
// мировая матрица, матрица вида, матрица проекции
//-----
VOID Matrix()
{
    // мировая матрица
    D3DXMATRIX MatrixWorld, MatrixWorldX;
    D3DXMATRIX MatrixWorldY, MatrixWorldZ;
    D3DXMATRIX MatrixView;           // матрица вида
    D3DXMATRIX MatrixProjection;     // матрица проекции
    // MatrixWorld
    UINT Time = timeGetTime() % 5000;
    FLOAT Angle = Time * (2.0f * D3DX_PI) / 5000.0f;
    D3DXMatrixRotationX(&MatrixWorldX, Angle);
    D3DXMatrixRotationY(&MatrixWorldY, Angle);
    D3DXMatrixMultiply(&MatrixWorld, &MatrixWorldX, &MatrixWorldY);
    pDirect3DDevice -> SetTransform(D3DTS_WORLD, &MatrixWorld);
    // MatrixView

```

```

D3DXMatrixLookAtLH(&MatrixView,
                    &D3DXVECTOR3(0.0f, 0.0f, -8.0f),
                    &D3DXVECTOR3(0.0f, 0.0f, 0.0f),
                    &D3DXVECTOR3(0.0f, 1.0f, 0.0f));
pDirect3DDevice -> SetTransform(D3DTS_VIEW, &MatrixView);
// MatrixProjection
D3DXMatrixPerspectiveFovLH(&MatrixProjection, D3DX_PI/4,
                           1.0f, 1.0f, 100.0f);
pDirect3DDevice -> SetTransform(D3DTS_PROJECTION,
                                &MatrixProjection);
}
//-----
//  Функция
//  RenderingDirect3D()
//  рисование
//-----
VOID RenderingDirect3D()
{
    if (pDirect3DDevice == NULL)        // проверяем ошибки
        return;
    pDirect3DDevice -> Clear(0, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
                            D3DCOLOR_XRGB(60, 100, 150), 1.0f, 0);

    // начало сцены
    pDirect3DDevice -> BeginScene();
    // здесь происходит прорисовка сцены
    Matrix();
    pDirect3DDevice -> SetStreamSource(0, pBufferVershin, 0,
                                       sizeof(CUSTOMVERTEX));

    pDirect3DDevice -> SetFVF(D3DFVF_CUSTOMVERTEX);
    pDirect3DDevice -> SetIndices(pBufferIndex);
    // вывод объекта
    pDirect3DDevice -> DrawIndexedPrimitive(D3DPT_TRIANGLELIST,
                                             0, 0, 36, 0, 12);

    // конец сцены
    pDirect3DDevice -> EndScene();
    pDirect3DDevice -> Present(NULL, NULL, NULL, NULL);
}

```

```

//-----
//  функция
//  DeleteDirect3D()
//  освобождает захваченные ресурсы
//-----
VOID DeleteDirect3D()
{

    if (pBufferIndex != NULL)
        pBufferIndex -> Release();

    if (pBufferVershin != NULL)
        pBufferVershin -> Release();

    if (pDirect3DDevice != NULL)
        pDirect3DDevice -> Release();

    if (pDirect3D != NULL)
        pDirect3D -> Release();
}

//-----
//  функция
//  MainWinProc()
//  здесь происходит обработка сообщений
//-----
LRESULT CALLBACK MainWinProc(HWND    hwnd,
                                UINT    msg,
                                WPARAM  wparam,
                                LPARAM  lparam)
{
    switch (msg)
    {
        case WM_DESTROY:
        {
            DeleteDirect3D();
            PostQuitMessage(0);
            return(0);
        }
    }
}

```

```

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

//-----
//  функция
//  WinMain
//  входная точка приложения
//-----

int WINAPI WinMain(HINSTANCE hinstance,
                  HINSTANCE hprevinstance,
                  LPSTR lpcmdline,
                  int ncmdshow)
{
    WNDCLASSEX windowclass;    // создаем класс
    HWND        hwnd;          // создаем дескриптор окна
    MSG         msg;           // идентификатор сообщения
    // определим класс окна WNDCLASSEX
    windowclass.cbSize = sizeof(WNDCLASSEX);
    windowclass.style = CS_DBLCLKS|CS_OWNDC|CS_HREDRAW|CS_VREDRAW;
    windowclass.lpfnWndProc = MainWinProc;
    windowclass.cbClsExtra = 0;
    windowclass.cbWndExtra = 0;
    windowclass.hInstance = hinstance;
    windowclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    windowclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    windowclass.hbrBackground = (HBRUSH)GetStockObject(GRAY_BRUSH);
    windowclass.lpszMenuName = NULL;
    windowclass.lpszClassName = "WINDOWSCCLASS";
    windowclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
    // регистрируем класс
    if (!RegisterClassEx(&windowclass))
        return(0);
    // теперь, когда класс зарегистрирован, можно создать окно
    if (!(hwnd = CreateWindowEx(NULL, "WINDOWSCCLASS", "Текст",
                               WS_OVERLAPPEDWINDOW|WS_VISIBLE, 300, 150,
                               500, 400, NULL, NULL, hinstance, NULL)))
        return(0);
    if (SUCCEEDED(InitialDirect3D(hwnd)))
    {
        if (SUCCEEDED(InitialObject() ))

```

```
{
    ShowWindow(hwnd, SW_SHOWDEFAULT);
    UpdateWindow(hwnd);
    ZeroMemory(&msg, sizeof(msg));
    while (msg.message != WM_QUIT)
    {
        if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else
            RenderingDirect3D();
    }
}

return 0;
}
```

Итоги урока

Буфер глубины или Z-буфер — был темой этого урока. Z-буфер занимается сортировкой вершин объекта в зависимости от камеры. Мы раскрасили куб в разные цвета и использовали матричную конкатенацию. *Урок 8* будет посвящен освещению и материалу объекта.

Урок 8



Свет и материал

На *уроке 7* был создан разноцветный куб, вращающийся по осям X и Y , имеющий свои локальные координаты и находящийся в центре нашего приложения. С помощью матричных функций куб можно вращать по осям, удалять или наоборот приближать в пространстве, задавать место, а точнее камеру просмотра сцены. Но при всем при этом сам куб вроде бы и был кубом, но выглядел как-то нереалистично. Для того чтобы более естественно отображать объекты на экране, в Direct3D определены понятия *материала* и *света*. Материал в Direct3D — это свойства поверхности рисуемого примитива. С помощью материала можно определить, как будет отражаться от поверхности 3D-объекта свет. Когда вы используете свет, вам обязательно нужно задействовать материал, и наоборот. Одно без другого работать не будет. Как уже говорилось, материал — это свойства поверхности объекта, сами свойства определены в структуре `D3DMATERIAL9`, где они и задаются. Рассмотрим прототип этой структуры:

```
tupedef struct _D3DMATERIAL9 {  
    D3DCOLORVALUE    Diffuse;  
    D3DCOLORVALUE    Ambient;  
    D3DCOLORVALUE    Specular;  
    D3DCOLORVALUE    Emissive;  
    FLOAT            Power;  
} D3DMATERIAL9;
```

Структура `D3DMATERIAL9` имеет следующие параметры:

- ☐ `Diffuse` — значение структуры `D3DCOLORVALUE`, определяет диффузный или рассеянный свет, исходящий от материала;
- ☐ `Ambient` — значение структуры `D3DCOLORVALUE`, определяет цвет подсветки, иногда еще говорят — окружающий свет;
- ☐ `Specular` — значение структуры `D3DCOLORVALUE`, определяет отражающий или зеркальный свет. С его помощью можно определить вид бликов;

- Emissive — значение структуры `D3DCOLORVALUE`, определяет эмиссионный или излучающий свет материала;
- Power — это размер, а скорее мощность отражения, определенная для материала. Когда необходимо отключить блики, значение выставляется в 0.

Параметры `Diffuse`, `Ambient`, `Specular` и `Emissive` являются значениями структуры `D3DCOLORVALUE`. Прототип этой структуры выглядит следующим образом:

```
typedef struct _D3DCOLORVALUE {  
    FLOAT r;  
    FLOAT g;  
    FLOAT b;  
    FLOAT a;  
} D3DCOLORVALUE;
```

Структура `D3DCOLORVALUE` имеет такие параметры:

- `r` — компонент, содержащий красный цвет;
- `g` — компонент, содержащий зеленый цвет;
- `b` — компонент, содержащий синий цвет;
- `a` — компонент, содержащий альфа-цвет, еще говорят альфа-канал, обеспечивающий прозрачность цвета для реалистического отображения, скажем, воды или стекла.

Изменяя все значения этой структуры от 0.0 до 1.0, задают разные цвета. Все значения выставлены в 0.0, соответствующее черному цвету. Если применить значение больше 1.0, то получаются более яркие цвета, а если отрицательные значения, то более темные.

Задавая параметры структуры `D3DMATERIAL9`, вы определяете свойства поверхности создаваемого 3D-объекта. Но как уже говорилось, материал без света не работает, можете для интереса поэкспериментировать на кубе. Все, что вы увидите на экране, — это вращающийся куб черного цвета. Также если попробовать применить свет без материала, результат будет аналогичным. Чтобы создать материал, необходимо объявить соответствующую переменную:

```
D3DMATERIAL9 Material;
```

и с помощью заполнения полей структуры `D3DMATERIAL9` определить свойства поверхности. Но об этом чуть позже, а сейчас давайте рассмотрим свойства света.

Свет

Свет в Direct3D — это прямая аналогия нашего мира, представляющая собой окружающее нас освещение. Чтобы увидеть созданный объект, в Direct3D необходимо задать источник света, что вполне логично, вы ведь ничего не увидите в каком-нибудь темном подвале, не имея источника освещения. Источники света бывают трех типов. *Параллельный* или *направленный* (directional) — этот тип не имеет определенного источника света, он как бы повсюду, но светит в одном направлении. *Точечный* (point-source) — тип, имеющий определенный источник света, но светящийся во всех направлениях. Например, обыкновенная лампочка, излучающая свет во все стороны. *Прожекторный* или *нацеленный* (spotlight) — тип, имеющий определенный источник света, но светящийся в заданном направлении в виде направленного конуса, например, фонарик, прожектор. Свет в Direct3D состоит из трех цветов, которые независимо друг от друга принимают участие в вычислениях используемого освещения. Все три цвета, а это разбросанный, окружающий и зеркальный цвет, напрямую взаимодействуют с тремя цветами, определенными в свойствах материала. Окружающий цвет освещения с окружающим материала, зеркальный с зеркальным. Результат такого сочетания и есть конечный цвет освещения сцены, а также конечный цвет, отражаемый материалом, определенный в свойствах поверхности примитива. Альфа-цвет в освещении не применяется, но может использоваться в свойствах материала. Для того чтобы определить все значения цветов, участвующих в освещении объекта, существует структура D3DLIGHT9. Рассмотрим прототип этой структуры:

```
typedef struct _D3DLIGHT9 {  
    D3DLIGHTTYPE    Type;  
    D3DCOLORVALUE    Diffuse;  
    D3DCOLORVALUE    Specular;  
    D3DCOLORVALUE    Ambient;  
    D3DVECTOR        Position;  
    D3DVECTOR        Direction;  
    FLOAT            Range;  
    FLOAT            Falloff;  
    FLOAT            Attenuation_0;  
    FLOAT            Attenuation_1;  
    FLOAT            Attenuation_2;  
    FLOAT            Theta;  
    FLOAT            Phi;  
} D3DLIGHT9;
```

Приведу параметры структуры `D3DLIGHT9`:

- ❑ `Type` — член перечисляемого типа `D3DLIGHTTYPE`. Это значение определяет тип источника света;
- ❑ `Diffuse` — диффузный или рассеянный свет, излучаемый источником света. Член структуры `D3DCOLORVALUE`;
- ❑ `Specular` — зеркальный свет, излучаемый источником света. Член структуры `D3DCOLORVALUE`;
- ❑ `Ambient` — окружающий свет, излучаемый источником света. Член структуры `D3DCOLORVALUE`;
- ❑ `Position` — в этом параметре с помощью структуры `D3DVECTOR` задаются координаты источника света в сцене. Когда используется направленный или параллельный (`directional`) источник света, этот параметр игнорируется, поскольку данный тип не имеет определенного источника света;
- ❑ `Direction` — задается структурой `D3DVECTOR` и определяет направление падающего света. Работает только с направленным (`directional`) и прожекторным (`spotlight`) типом источника света;
- ❑ `Range` — максимальное расстояние для освещения рисуемой сцены;
- ❑ `Falloff` — этот параметр уменьшает освещение между внутренним конусом, определенным как угол `Theta`, и внешним углом `Phi` источника света. Например, как в фонарике — большой круг света, внутри которого меньший круг от лампы, более ярко светящийся. Значение по умолчанию 1.0 для более равномерного перехода от двух конусов. С этой величиной интересно поэкспериментировать;
- ❑ `Attenuation_0`, `Attenuation_1` и `Attenuation_2` — эти три параметра влияют на интенсивность освещения за счет изменения расстояния от объекта до источника света;
- ❑ `Theta` — угол внутреннего конуса источника света, измеряется в радианах. Минимальное значение 0, максимальное определено в параметре `Phi`;
- ❑ `Phi` — угол внешнего конуса источника света, измеряется в радианах. Минимальное значение 0, максимальное — константа π . За пределами этого конуса освещения нет.

Первый параметр структуры `D3DLIGHT9`, `Type` — член перечисляемого типа `D3DLIGHTTYPE`, определяющий источник света, нуждается в отдельном рассмотрении. Рассмотрим прототип структуры `D3DLIGHTTYPE`, с помощью которой определяется тип источника света:

```
typedef enum _D3DLIGHTTYPE {
    D3DLIGHT_POINT          = 1,
    D3DLIGHT_SPOT           = 2,
```

```
D3DLIGHT_DIRECTIONAL = 3,  
D3DLIGHT_FORCE_DWORD = 0x7fffffff,  
} D3DLIGHTTYPE;
```

Структура `D3DLIGHTTYPE` имеет следующие параметры:

- ❑ `D3DLIGHT_POINT` — точечный источник света, разбрасывающий свет во все стороны. Этот источник рассчитывается немного медленнее, но отображается лучше;
- ❑ `D3DLIGHT_SPOT` — прожекторный или нацеленный источник света, светящийся в определенном направлении в виде конуса;
- ❑ `D3DLIGHT_DIRECTIONAL` — параллельный или направленный свет, не имеющий источника света, но имеющий направления. Может распространяться до бесконечности;
- ❑ `D3DLIGHT_FORCE_DWORD` — не используется.

Все три константы, установленные в данной структуре, определяют типы источника света, рассмотренные в начале урока.

Чтобы создать освещение, необходимо объявить переменную:

```
D3DLIGHT9 Light;
```

и заполнить нужные вам для работы поля структуры `D3DLIGHT9`.

Нормаль

Всегда, когда используются материал и освещение, необходимо определить нормаль. *Нормаль* — это вектор, расположенный перпендикулярно одной из сторон рисуемого примитива (рис. 8.1).

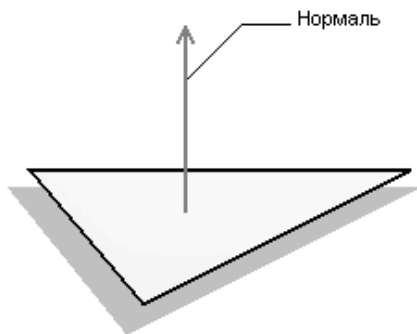


Рис. 8.1. Нормаль

Нормаль помогает точно рассчитать освещенность объекта относительно выбранного источника света. Она может назначаться для каждой вершины рисуемого примитива, в нашем примере куба, посмотрите на рис. 8.2.

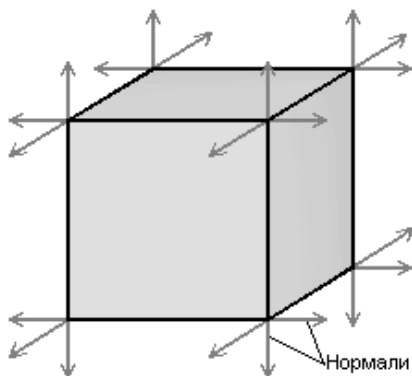


Рис. 8.2. Направление нормалей для куба

У лицевой стороны куба нормали для четырех вершин направлены перпендикулярно всей отображаемой стороне, т. е. по оси Z от куба в направлении камеры или наших глаз. Нормали для левой и правой сторон куба так же направлены перпендикулярно влево и вправо по оси X . Аналогично направление нормалей можно найти для всех вершин верхней и нижней стороны куба.

Установив нормали для всего объекта (о способе задания нормалей чуть позже), вы тем самым определите, как куб будет отображаться в сцене, когда на него будет направлен источник света. Источник света в примере будет направленный или параллельный от наших глаз в сторону куба, освещающий все на своем пути до бесконечности, которую, правда, зададим мы сами в параметре `Range` структуры `D3DLIGHT9`. При вращении куба в заданном ранее направлении, как только какие-либо из нормалей попадут в фокус источника освещения, `Direct3D` произведет автоматический расчет на основании находящихся в фокусе нормалей и весь объект будет освещаться, а не попавшие нормали, или точнее, стороны куба, затенятся. Такой своего рода механизм установки нормалей, позволяет производить расчет правильного освещения рисуемого объекта. Величина самой нормали, по требованию `Direct3D`, должна быть нормализована и иметь единичную длину, т. е. задаваться значением 1.0.

Все нормали определяются в структуре `CUSTOMVERTEX`. Вместо значения `color`, с помощью которого мы закрашивали куб, даже можно сказать освещали его своими внутренними средствами, появятся три значения: `nx`, `ny` и

nz для каждой вершины куба. Итак, мы сейчас подошли к рассмотрению кода для этого урока. Он находится на компакт-диске в одноименной папке Urok8. Создайте свой проект и займемся программированием. Скопируйте код с предыдущего урока, находящийся на компакт-диске в папке Code\Urok8, в новый файл LightMaterial.cpp и начнем.

В глобальных переменных изменяем структуру CUSTOMVERTEX:

```
struct CUSTOMVERTEX
{
    FLOAT X, Y, Z;           // координаты вершин
    FLOAT nx, ny, nz;        // нормали
};
```

В структуре добавляются координаты, с помощью которых будет определяться направление нормали для каждой вершины. Еще необходимо определить новый формат вершин. Поскольку значение `color` уже не используется, то флаг `D3DFVF_DIFFUSE` становится недействительным. Для определения нового формата вершин, использующих нормали, назначается флаг `D3DFVF_NORMAL`, показывающий, что будут использоваться нормали для расчета освещения сцены:

```
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ | D3DFVF_NORMAL)
```

Следующая за этими строками функция `InitialDirect3D()` остается без изменения. В функции `InitialObject()` изменится формат записи структуры `CUSTOMVERTEX Vershin`. Как и прежде, первые три значения являются координатами вершин куба, а следующие три значения будут использоваться для задания нормалей всех вершин:

```
CUSTOMVERTEX Vershin[] =
{
    {1.0f, -1.0f, -1.0f, 0.0f, 0.0f, -1.0f},    // A
    {1.0f, 1.0f, -1.0f, 0.0f, 0.0f, -1.0f},     // B
    {-1.0f, 1.0f, -1.0f, 0.0f, 0.0f, -1.0f},    // C
    {-1.0f, -1.0f, -1.0f, 0.0f, 0.0f, -1.0f},   // D

    {-1.0f, -1.0f, -1.0f, -1.0f, 0.0f, 0.0f},   // A2
    {-1.0f, 1.0f, -1.0f, -1.0f, 0.0f, 0.0f},    // B2
    {-1.0f, 1.0f, 1.0f, -1.0f, 0.0f, 0.0f},     // C2
    {-1.0f, -1.0f, 1.0f, -1.0f, 0.0f, 0.0f},    // D2

    {-1.0f, -1.0f, 1.0f, 0.0f, 0.0f, 1.0f},     // A3
    {-1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f},      // B3
```

```

{1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f},      // C3
{1.0f, -1.0f, 1.0f, 0.0f, 0.0f, 1.0f},     // D3

{1.0f, -1.0f, 1.0f, 1.0f, 0.0f, 0.0f},     // A4
{1.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f},     // B4
{1.0f, 1.0f, -1.0f, 1.0f, 0.0f, 0.0f},     // C4
{1.0f, -1.0f, -1.0f, 1.0f, 0.0f, 0.0f},    // D4

{1.0f, -1.0f, -1.0f, 0.0f, -1.0f, 0.0f},   // A5
{-1.0f, -1.0f, -1.0f, 0.0f, -1.0f, 0.0f}, // B5
{-1.0f, -1.0f, 1.0f, 0.0f, -1.0f, 0.0f},   // C5
{1.0f, -1.0f, 1.0f, 0.0f, -1.0f, 0.0f},    // D5

{1.0f, 1.0f, 1.0f, 0.0f, 1.0f, 0.0f},      // A6
{-1.0f, 1.0f, 1.0f, 0.0f, 1.0f, 0.0f},     // B6
{-1.0f, 1.0f, -1.0f, 0.0f, 1.0f, 0.0f},    // C6
{1.0f, 1.0f, -1.0f, 0.0f, 1.0f, 0.0f},     // D6
};

```

Как вы уже знаете, нормаль необходимо нормализовать и задавать единичным значением. Для первой стороны куба лицевые нормали задаются по оси Z и направлены перпендикулярно лицевой стороне куба, а значит, в направлении камеры, поэтому значение ставится $-1.0f$ по Z . Следующая сторона примитива — левая часть. Все четыре нормали для каждой вершины так же расположены перпендикулярно стороне куба и направлены в сторону по оси X , а значит, равны $-1.0f$. То же самое и для правой части куба, с той лишь разницей, что n_x примет значение $1.0f$, поскольку нормали направлены в положительную сторону оси X . То есть вы задаете направление каждой нормали с помощью одного из значений n_x , n_y и n_z для каждой вершины. Для нижней и верхней сторон куба ситуация аналогичная. Все три компонента: материал, свет и нормаль очень жестко взаимодействуют между собой, как бы цепляясь друг за друга, поэтому ошибка в одном месте скажется на всем результате.

Функция `Matrix()` остается без изменения. Сразу за ней мы создадим новую функцию `LightMaterial()`, в которой будет происходить работа материала и света.

Установка света и материала

Первым делом объявим две переменные для материала и освещения:

```
D3DMATERIAL9 Material;  
D3DLIGHT9 Light;
```

Далее установим материал для поверхности куба:

```
ZeroMemory(&Material, sizeof(D3DMATERIAL9));  
Material.Diffuse.r = Material.Ambient.r = 1.0f;  
Material.Diffuse.g = Material.Ambient.g = 1.0f;  
Material.Diffuse.b = Material.Ambient.b = 0.0f;  
Material.Diffuse.a = Material.Ambient.a = 1.0f;  
pDirect3DDevice -> SetMaterial(&Material);
```

В такой записи происходит присвоение двух цветов, диффузного и цвета подсветки, а с помощью комбинаций значений для ARGB задается желтый цвет материала. Для одного объекта назначается один материал, если вы попытаетесь нарисовать два объекта, скажем два куба, то для каждого объекта сначала создается материал, например:

```
D3DMATERIAL9 Material1, Material2;  
далее устанавливается сам материал путем заполнения структуры D3DMATERIAL9.  
После чего с помощью функции IDirect3DDevice9::SetMaterial устанавли-  
ваем свойства материала для рисуемого объекта. Прототип функции выглядит  
достаточно просто:
```

```
HRESULT SetMaterial(CONST D3DMATERIAL9 *pMaterial);
```

Функция `SetMaterial()` имеет единственный параметр `pMaterial` — указатель на структуру `D3DMATERIAL9`, результат всей операции по установке свойств поверхности объекта.

Такими вот незатейливыми действиями создается материал. Мы задали желтый цвет поверхности объекта и теперь приступим к освещению куба. Установим освещение сцены. Самих источников может быть много, но каждый несколько замедляет работу сцены.

Заполним структуру `D3DLIGHT9`:

```
Light.Type = D3DLIGHT_DIRECTIONAL;
```

Так определяется тип источника света, в этом примере он направленный, т. е. не имеющий фиксированного источника, но распространяющийся в заданном направлении.

С помощью значений RGB задан белый свет, обычно это свет по умолчанию:

```
Light.Diffuse.r = 1.0f;
Light.Diffuse.g = 1.0f;
Light.Diffuse.b = 1.0f;
```

Но когда вы, например, создаете световую компоненту для камина, он может быть красным или оранжевым, на усмотрение программиста. В строке

```
Light.Range = 1000.0f;
```

устанавливается максимальное расстояние, на которое источник света будет действовать. После чего необходимо определить значение для инициализации света, а именно вектор для определения направления источника света:

```
VectorDir = D3DXVECTOR3(0.0f, 0.0f, 1.0f),
```

Всю сцену мы наблюдаем со стороны оси *Z*. Для того чтобы направление света было параллельно нашему взгляду, задается такое значение: (0, 0, 1). Мы смотрим в центр куба, и наш взгляд направлен в положительную сторону оси *Z*. Если поменять значение 1 на -1, то свет будет светить в нашу сторону, но поскольку он находится за кубом, то его просто не будет видно. В этом случае, чтобы увидеть результат освещения, достаточно задать другие значения для *VectorDir* по осям *X* и *Y* для смещения источника света в любую из сторон.

Далее с помощью функции `D3DXVec3Normalize()` устанавливается нормаль для определенного вектора *VectorDir*. Прототип функции выглядит следующим образом:

```
D3DXVECTOR3* D3DXVec3Normalize(
    D3DXVECTOR3*    pOut;
    CONST D3DXVECTOR* pV);
```

Функция `D3DXVec3Normalize()` имеет такие параметры:

- *pOut* — указатель на структуру `D3DXVECTOR3`, выходной результат всей операции;
- *pV* — указатель на структуру `D3DXVECTOR`, исходное значение.

Установим нормаль для вектора *VectorDir*:

```
D3DXVec3Normalize((D3DXVECTOR3*)&Light.Direction, &VectorDir);
```

После этих действий необходимо назначить устройству `Direct3D` созданный нами источник света путем вызова функции `IDirect3DDevice9::SetLight`. Ее прототип имеет такую запись:

```
HRESULT SetLight(
    DWORD          Index,
    CONST D3LIGHT9* pLight);
```

Функция `SetLight()` включает следующие параметры:

- ❑ `Index` — это индекс, служит для установления свойств света. Отсчет начинается с 0;
- ❑ `pLight` — это и есть созданный источник света, указатель на структуру `D3DLIGHT9`.

Зная все значения, можно записать:

```
pDirect3DDevice -> SetLight(0, &Light);
```

После этого необходимо вызвать функцию `IDirect3DDevice9::LightEnable`, с ее помощью вы можете разрешить или запретить использование созданных свойств для источника света. Ее прототип имеет такую запись:

```
HRESULT LightEnable(  
    DWORD LightIndex,  
    BOOL bEnable);
```

Функция `LightEnable()` включает следующие параметры:

- ❑ `LightIndex` — в этом параметре указывается индекс, определяющий свойства света. Отсчет идет от 0;
- ❑ `bEnable` — булева константа, имеющая два значения `FALSE` и `TRUE`, соответственно запрещающая и разрешающая использовать свет в значении `LightIndex`.

В этой и предыдущей функции ситуация с индексами выглядит несколько запутанной, но на самом деле все просто. Источников света может быть более одного, и по индексам определяется, какому именно источнику света назначены те или иные действия. Сразу же возникает вопрос, а вдруг параметры индекса будут заданы не верно? Тогда свойства света задаются по умолчанию заранее определенными значениями для структуры `D3DLIGHT9`:

```
Light.Type           = D3DLIGHT_DIRECTIONAL;  
Light.Diffuse.r      = 1.0f;  
Light.Diffuse.g      = 1.0f;  
Light.Diffuse.b      = 1.0f;  
Light.Specular.r     = 0.0f;  
Light.Specular.g     = 0.0f;  
Light.Specular.b     = 0.0f;  
Light.Ambient.r      = 0.0f;  
Light.Ambient.g      = 0.0f;  
Light.Ambient.b      = 0.0f;  
Light.Direction     = Vector;
```

где `Vector` — это указатель на структуру `D3DXVECTOR3`, имеющий значения (0,0,1). Все остальные параметры структуры `D3DLIGHT9` выставляются в 0.

Давайте разрешим использование всех определенных свойства света для нашего направленного источника:

```
pDirect3DDevice -> LightEnable(0, TRUE);
```

После чего следует двойной вызов функции

```
IDirect3DDevice9::SetRenderState:
```

```
pDirect3DDevice -> SetRenderState(D3DRS_LIGHTING, TRUE);
```

```
pDirect3DDevice -> SetRenderState(D3DRS_AMBIENT, 0);
```

В первом случае мы просто перенесли вызов из функции `InitialDirect3D()` с *урока 7* в новое место, поскольку у нас теперь есть своя функция для работы с освещением. Вызываем ее со значением `TRUE`, тем самым разрешая работу света. И вторым вызовом этой функции определим дополнительное свойство отображения, используя флаг `D3DRS_AMBIENT`, включающий окружающее освещение. Второй параметр определяет работу первого, значение 0 по умолчанию.

На этом создание функции `LightMaterial()` заканчивается, и саму функцию вызовем в `RenderingDirect3D()`, в месте, где происходит прорисовка сцены.

Все, что в итоге получилось, можно увидеть в листинге 8.1.

Листинг 8.1. Файл `LightMaterial.cpp`

```
//-----
//  LightMaterial.cpp
//  используем освещение и материал
//-----
#include <windows.h>    // подключаем заголовочный файл Windows
#include <d3d9.h>        // подключаем заголовочный файл DirectX 9 SDK
#include <d3dx9.h>       // подключаем библиотеку D3DX-утилит
#include <mmsystem.h>    // подключаем системную библиотеку
//-----
//  глобальные переменные
//-----
LPDIRECT3D9 pDirect3D = NULL;           // 3D-объект
LPDIRECT3DDEVICE9 pDirect3DDevice = NULL; // устройство
LPDIRECT3DVERTEXBUFFER9 pBufferVershin = NULL; // буфер вершин
LPDIRECT3DINDEXBUFFER9 pBufferIndex = NULL; // индексный буфер
```

```

struct CUSTOMVERTEX
{
    FLOAT X, Y, Z;           // координаты вершины
    FLOAT nx, ny, nz;        // нормали
};

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_NORMAL) // формат вершины
//-----
// функция
// IDirect3D()
// инициализация Direct3D
//-----
HRESULT InitializeDirect3D(HWND hwnd)
{
    if (NULL == (pDirect3D = Direct3DCreate9(D3D_SDK_VERSION)))
        return E_FAIL;
    D3DDISPLAYMODE Display;
    if (FAILED(pDirect3D -> GetAdapterDisplayMode(
        D3DADAPTER_DEFAULT, &Display)))
        return E_FAIL;
    D3DPRESENT_PARAMETERS Direct3DParametr;
    ZeroMemory(&Direct3DParametr, sizeof(Direct3DParametr));
    Direct3DParametr.Windowed = TRUE;
    Direct3DParametr.SwapEffect = D3DSWAPEFFECT_DISCARD;
    Direct3DParametr.BackBufferFormat = Display.Format;
    Direct3DParametr.EnableAutoDepthStencil = TRUE;
    Direct3DParametr.AutoDepthStencilFormat = D3DFMT_D16;
    if (FAILED(pDirect3D -> CreateDevice(D3DADAPTER_DEFAULT,
        D3DDEVTYPE_HAL, hwnd, D3DCREATE_SOFTWARE_VERTEXPROCESSING,
        &Direct3DParametr, &pDirect3DDevice)))
        return E_FAIL;
    // включаем отсечение Direct3D
    pDirect3DDevice -> SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);
    // подключаем Z-буфер
    pDirect3DDevice -> SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
    return S_OK;
}
//-----
// функция
// InitializeObject()

```

```
// создание объекта
//-----
HRESULT InitialObject()
{
    CUSTOMVERTEX Vershin[] =
    {
        {1.0f, -1.0f, -1.0f, 0.0f, 0.0f, -1.0f},    // A
        {1.0f, 1.0f, -1.0f, 0.0f, 0.0f, -1.0f},    // B
        {-1.0f, 1.0f, -1.0f, 0.0f, 0.0f, -1.0f},    // C
        {-1.0f, -1.0f, -1.0f, 0.0f, 0.0f, -1.0f},   // D

        {-1.0f, -1.0f, -1.0f, -1.0f, 0.0f, 0.0f},   // A2
        {-1.0f, 1.0f, -1.0f, -1.0f, 0.0f, 0.0f},    // B2
        {-1.0f, 1.0f, 1.0f, -1.0f, 0.0f, 0.0f},     // C2
        {-1.0f, -1.0f, 1.0f, -1.0f, 0.0f, 0.0f},    // D2

        {-1.0f, -1.0f, 1.0f, 0.0f, 0.0f, 1.0f},     // A3
        {-1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f},      // B3
        {1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f},       // C3
        {1.0f, -1.0f, 1.0f, 0.0f, 0.0f, 1.0f},      // D3

        {1.0f, -1.0f, 1.0f, 1.0f, 0.0f, 0.0f},     // A4
        {1.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f},      // B4
        {1.0f, 1.0f, -1.0f, 1.0f, 0.0f, 0.0f},     // C4
        {1.0f, -1.0f, -1.0f, 1.0f, 0.0f, 0.0f},    // D4

        {1.0f, -1.0f, -1.0f, 0.0f, -1.0f, 0.0f},   // A5
        {-1.0f, -1.0f, -1.0f, 0.0f, -1.0f, 0.0f},   // B5
        {-1.0f, -1.0f, 1.0f, 0.0f, -1.0f, 0.0f},    // C5
        {1.0f, -1.0f, 1.0f, 0.0f, -1.0f, 0.0f},     // D5

        {1.0f, 1.0f, 1.0f, 0.0f, 1.0f, 0.0f},      // A6
        {-1.0f, 1.0f, 1.0f, 0.0f, 1.0f, 0.0f},      // B6
        {-1.0f, 1.0f, -1.0f, 0.0f, 1.0f, 0.0f},     // C6
        {1.0f, 1.0f, -1.0f, 0.0f, 1.0f, 0.0f},     // D6
    };

    const unsigned short Index[]={
        0,1,2,      2,3,0,
        4,5,6,      6,7,4,
```

```

    8,9,10,    10,11,8,
    12,13,14,  14,15,12,
    16,17,18,  18,19,16,
    20,21,22,  22,23,20,
};

// создаем буфер вершин
if (FAILED(pDirect3DDevice -> CreateVertexBuffer(
    36*sizeof(CUSTOMVERTEX), 0, D3DFVF_CUSTOMVERTEX,
    D3DPOOL_DEFAULT, &pBufferVershin, NULL)))

return E_FAIL;
// блокируем
VOID* pBV;
if (FAILED(pBufferVershin->Lock(0, sizeof(Vershin), (void**)&pBV, 0)))
return E_FAIL;
// копируем
memcpy(pBV, Vershin, sizeof(Vershin));
// разблокируем
pBufferVershin -> Unlock();
// создаем индексный буфер
pDirect3DDevice -> CreateIndexBuffer(36*sizeof(Index), 0,
    D3DFMT_INDEX16, D3DPOOL_DEFAULT, &pBufferIndex, NULL);
// блокируем
VOID* pBI;
pBufferIndex -> Lock(0, sizeof(Index), (void**)&pBI, 0);
// копируем
memcpy(pBI, Index, sizeof(Index));
// разблокируем
pBufferIndex -> Unlock();
return S_OK;
}

//-----
// функция
// Matrix()
// мировая матрица, матрица вида, матрица проекции
//-----
VOID Matrix()
{
    D3DXMATRIX MatrixWorld, MatrixWorldX, MatrixWorldY;
    D3DXMATRIX MatrixView;          // матрица вида

```

```

D3DXMATRIX MatrixProjection; // матрица проекции
// MatrixWorld
UINT Time = timeGetTime() % 5000;
FLOAT Angle = Time * (2.0f * D3DX_PI) / 5000.0f;
D3DXMatrixRotationX(&MatrixWorldX, Angle);
D3DXMatrixRotationY(&MatrixWorldY, Angle);
D3DXMatrixMultiply(&MatrixWorld, &MatrixWorldX, &MatrixWorldY);
pDirect3DDevice -> SetTransform(D3DTS_WORLD, &MatrixWorld);
// MatrixView
D3DXMatrixLookAtLH(&MatrixView, &D3DXVECTOR3(0.0f, 0.0f, -8.0f),
                  &D3DXVECTOR3(0.0f, 0.0f, 0.0f),
                  &D3DXVECTOR3(0.0f, 1.0f, 0.0f));
pDirect3DDevice -> SetTransform(D3DTS_VIEW, &MatrixView);
// MatrixProjection
D3DXMatrixPerspectiveFovLH(&MatrixProjection,
                          D3DX_PI/4, 1.0f, 1.0f, 100.0f);
pDirect3DDevice -> SetTransform(D3DTS_PROJECTION, &MatrixProjection);
}
//-----
// функция
// LightMaterial()
// инициализация света и материала
//-----
VOID LightMaterial()
{
    D3DMATERIAL9 Material; // материал
    D3DLIGHT9 Light; // свет
    // установим материал
    ZeroMemory(&Material, sizeof(D3DMATERIAL9));
    Material.Diffuse.r = Material.Ambient.r = 1.0f;
    Material.Diffuse.g = Material.Ambient.g = 1.0f;
    Material.Diffuse.b = Material.Ambient.b = 0.0f;
    Material.Diffuse.a = Material.Ambient.a = 1.0f;
    pDirect3DDevice -> SetMaterial(&Material);
    D3DXVECTOR3 VectorDir;
    // установим свет
    ZeroMemory(&Light, sizeof(D3DLIGHT9));
    Light.Type = D3DLIGHT_DIRECTIONAL;

```

[illegible]

```

    // конец сцены
    pDirect3DDevice -> EndScene();
    pDirect3DDevice -> Present(NULL, NULL, NULL, NULL);
}
//-----
//  Функция
//  DeleteDirect3D()
//  освобождение захваченных ресурсов
//-----
VOID DeleteDirect3D()
{
    if (pBufferIndex != NULL)
        pBufferIndex -> Release();

    if (pBufferVershin != NULL)
        pBufferVershin -> Release();

    if (pDirect3DDevice != NULL)
        pDirect3DDevice -> Release();

    if (pDirect3D != NULL)
        pDirect3D -> Release();
}
//-----
//  функция
//  MainWinProc()
//  здесь происходит обработка сообщений
//-----
LRESULT CALLBACK MainWinProc(HWND    hwnd,
                                UINT   msg,
                                WPARAM wparam,
                                LPARAM lparam)
{
    switch(msg)
    {
        case WM_DESTROY:
        {
            DeleteDirect3D();

```

[illegible]

```
return 0;
if (SUCCEEDED(InitialDirect3D(hwnd)))
{
    if (SUCCEEDED(InitialObject() ))
    {
        ShowWindow(hwnd, SW_SHOWDEFAULT);
        UpdateWindow(hwnd);
        ZeroMemory(&msg, sizeof(msg));
        while (msg.message != WM_QUIT)
        {
            if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
            {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
            }
            else
                RenderingDirect3D();
        }
    }
}
return 0;
}
```

Итоги урока

Этот урок был посвящен материалу и свету. Вы познакомились со способами задания материала и света, узнали о различных источниках света и о том, как материал и свет взаимосвязаны. Для расчета освещения были использованы нормали. На *уроке 9* мы займемся выводом текста на экран монитора.

Урок 9



Текст в Direct3D

На этом уроке будет рассмотрена важная тема — вывод текста на экран. В любой игре, какой бы маленькой она ни была, неизменным атрибутом являются различные диалоги или просто числовые значения, ведущие подсчет очков или, скажем, количество оставшихся патронов в пулемете. Все эти действия — обыкновенное следствие вызова нескольких функций Win32 API, интегрированных в приложение Direct3D. Сам алгоритм вывода текста весьма прост: сначала создается шрифт, потом он инициализируется, определяется область вывода текста на экран и в конце происходит прорисовка заданного текста. Ничего сложного и экстраординарного в этом нет. Единственное, что хотелось бы уточнить, это то, что все-таки в программировании игр обычно создается класс `Font`, отвечающий за инициализацию и вывод текста на экран, но в нашем случае за эти действия будет отвечать функция `DrawMyText()`, которую мы создадим сами. Сейчас вы только учитесь и познаете азы программирования для DirectX, но при создании класса для определения шрифта, а также для всех других составляющих, пройденных нами на предыдущих уроках (все, что было изучено можно тоже распределить в классы), необходимо было бы создать дополнительные файлы, отвечающие за создание класса и его реализацию, что может повлечь за собой путаницу. Предлагаемый пример в данный момент имеет маленький объем, и создание дополнительных классов не придаст ему большей гибкости. Но на последнем *уроке 19* весь изученный код мы разобьем на классы, отражающие наиболее реальное программирование игр под Windows. Тем более, что сам язык программирования C++ объектно-ориентированный и все его лучшие качества как раз и заключаются в структуре создания классов. Конечно, до создания графического "движка" нам еще далеко, да и цель книги не в этом, но мы на правильном пути.

Создание шрифта

Итак, для начала создадим пустой проект, по традиции назвав его `Urok9`, и добавим пустой файл, назвав его `Text.cpp`. Скопируйте в этот файл код с предыдущего урока (см. листинг 8.1) и начнем программирование.

В начало кода добавим подключение заголовочного файла, отвечающего за работу со шрифтом:

```
#include <d3dx9core.h>
```

После чего в глобальных переменных объявим три переменные:

```
LPD3DXFONT pFont = NULL;
```

```
RECT Rec;
```

```
HFONT hFont;
```

В первой строке `pFont` является указателем на интерфейс `ID3DXFont`, отвечающий за работу со шрифтом. Как обычно, всем указателям на интерфейсы DirectX присваивается значение `NULL`. Переменная `Rec` — член структуры `RECT`. С ее помощью задается область вывода текста на экран. Структура `RECT` определяет координаты области вывода текста, а точнее, создает невидимый прямоугольник, задавая координаты для левого верхнего угла прямоугольника и правого нижнего. Внутри прямоугольника будет выводиться текст. На рис. 9.1 это очень хорошо видно.

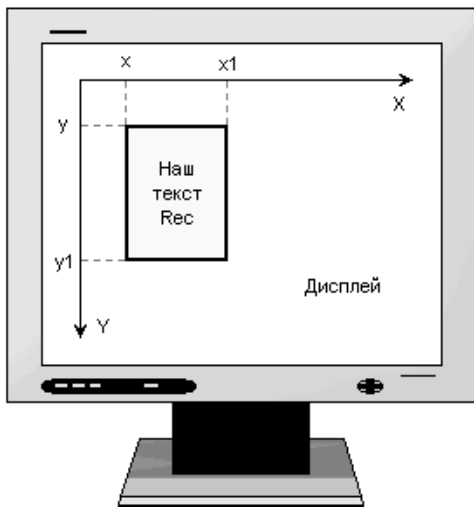


Рис. 9.1. Область вывода текста на экран

Рассмотрим прототип структуры `RECT`:

```
typedef struct _RECT  
{  
    LONG left;  
    LONG top;
```

```
    LONG right;  
    LONG bottom;  
} RECT, *PRECT;
```

Структура `RECT` включает следующие параметры:

- `left` — координата по оси X левого верхнего угла прямоугольника;
- `top` — координата по оси Y левого верхнего угла прямоугольника;
- `right` — координата по оси X правого нижнего угла;
- `bottom` — координата по оси Y правого нижнего угла.

Переменная `hFont` — это переменная структуры `HFONT`, содержащая шрифт, создаваемый с помощью функции `CreateFont()`. Вызов функции `CreateFont()` будет осуществляться в функции `DrawMyText()`, которую мы теперь непосредственно и рассмотрим. Предлагаю сначала посмотреть на реализацию всей функции, а потом в ней разобраться:

```
VOID DrawMyText(LPDIRECT3DDEVICE9 pDirect3DDevice, HFONT hFont,  
               char* StrokaTexta, int x, int y,  
               int x1, int y1, D3DCOLOR MyColor)  
{  
    // создаем шрифт  
    hFont = CreateFont(30, 10, 0, 0, FW_NORMAL, FALSE, FALSE, 0, 1,  
                      0, 0, 0, DEFAULT_PITCH|FF_MODERN, "Arial");  
    // координаты прямоугольника  
    Rec.left   = x;  
    Rec.top    = y;  
    Rec.right  = x1;  
    Rec.bottom = y1;  
    // инициализируем шрифт  
    D3DXCreateFont(pDirect3DDevice, hFont, &pFont);  
    // начало  
    pFont -> Begin();  
    // вывод текста  
    pFont -> DrawText(StrokaTexta, -1, &Rec, DT_WORDBREAK, MyColor);  
    // конец  
    pFont -> End();  
}
```

Функция `DrawMyText()` определена с восемью параметрами. С помощью различных манипуляций с этими параметрами происходит работа со значениями внутри функции `DrawMyText()`.

Первым действием функции будет создание шрифта. Сама библиотека DirectX понятия не имеет ни о каком шрифте, поэтому его необходимо создать с помощью функции `CreateFont()` и присвоить итоговые значения переменной `hFont`.

Прежде чем совершить эти действия, рассмотрим прототип функции `CreateFont()`:

```
HFONT CreateFont(  
    int      nHeight,  
    int      nWight,  
    int      nEscapement,  
    int      nOrientation,  
    int      fnWeight,  
    DWORD    fdwItalic,  
    DWORD    fdwUnderline,  
    DWORD    fdwStrikeOut,  
    DWORD    fdwCharSet,  
    DWORD    fdwOutputPrecision,  
    DWORD    fdwClipPrecision,  
    DWORD    fdwQuality,  
    DWORD    fdwPitchAndFamily,  
    LPCTSTR  LpszFase);
```

Функция `CreateFont()` включает следующие параметры:

- ❑ `nHeiht` — высота шрифта, выраженная в логических единицах. У нас высота шрифта 30 единиц;
- ❑ `nWight` — ширина шрифта, выраженная в логических единицах. Наша ширина 10 единиц;
- ❑ `nEscapement` — угол смещения или угол наклона всей строки текста в градусах;
- ❑ `nOrientation` — угол ориентации или угол наклона всей оси строки текста в градусах, например, для установки угла наклона текста типа курсив. Два параметра, определяющих углы смещения, мы не используем, поэтому установим значение 0 в обоих случаях;
- ❑ `fnWeight` — это "вес" шрифта. Здесь предусмотрено большое количество флагов, но у нас будет значение по умолчанию `FW_NORMAL`;
- ❑ `fdwItalic` — курсив шрифта. Игнорируем это значение — `FALSE`;
- ❑ `fdwUnderline` — подчеркивание шрифта. Ставим `FALSE`, значит, не используем;

- ❑ `fdwStrikeOut` — перечеркнутый шрифт. Мы не пользуемся этим значением, поэтому ставим 0;
- ❑ `fdwCharSet` — кодировка шрифта. Для задания значения используется одна из констант, определяющих кодировку шрифта. Чтобы корректно отображать русские буквы, поставьте 1;
- ❑ `fdwOutputPrecision` — точность вывода текста;
- ❑ `fdwClipPrecision` — точность обрезки текста;
- ❑ `fdwQuality` — качество вывода текста;
- ❑ `fdwPitchAndFamily` — в этом параметре определяются гарнитура и тип создаваемого шрифта. Чтобы определить значения для данного параметра, необходимо скомбинировать флаги с помощью побитового ИЛИ "|". Константы `DEFAULT_PITCH`, `FIXED_PITCH`, `VARIABLE_PITCH` определяют гарнитуру, а константы `FF_DECORATIVE`, `FF_DONTCARE`, `FF_MODERN`, `FF_ROMAN`, `FF_SCRIPT`, `FF_SWISS` зададут тип шрифта. Никаких ограничений в комбинациях не существует, задавайте, что угодно;
- ❑ `LpszFace` — последний параметр, обозначает имя шрифта. У нас распространенный Arial, но ничто не мешает вам выбрать Times New Roman, Courier New либо что-то другое.

Три параметра, отвечающие за точность вывода, обрезки и качество вывода текста, имеют огромное количество всевозможных констант, но мы воспользуемся значением по умолчанию, т. е. 0. Иначе мы утонем в Win32 API, а книга из уроков по DirectX превратится в уроки по Win32 API.

Задав, таким образом, значения для параметров функции `CreateFont()`, вы создадите шрифт, сохраняющийся в переменной `hFont`. После чего необходимо инициализировать созданный шрифт с помощью функции `D3DXCreateFont()`, прототип которой выглядит так:

```
HRESULT D3DXCreateFont(  
    LPDIRECT3DDEVICE9 pDevice,  
    HFONT              hFont,  
    LPD3DXFONT**       ppFont);
```

Параметры функции `D3DXCreateFont()` следующие:

- ❑ `pDevice` — это созданное нами устройство `pDirect3DDevice`, показывающее, что инициализированный шрифт принадлежит устройству Direct3D;
- ❑ `hFont` — переменная структуры `HFONT`, содержащая созданный ранее шрифт;
- ❑ `ppFont` — указатель на указатель `pFont`, содержащий результат инициализации шрифта.

Вывод текста на экран

После создания и инициализации шрифта надо определить область вывода текста на экране. Делается это с помощью задания значений для структуры `RECT`:

```
Rec.left    = x;
Rec.top     = y;
Rec.right   = x1;
Rec.bottom  = y1;
```

Передавая любые числовые значения для (x,y) и $(x1,y1)$ в функцию `DrawMyText()`, мы будем определять координаты прямоугольника. Кстати, ничто не мешает стать ему квадратом для вывода текста внутри этой области. В конце создаваемой функции `DrawMyText()` происходят вызовы функций `ID3DXFont::Begin` и `ID3DXFont::End`. Вам это ничего не напоминает? Именно функция `Begin()` — своего рода индикатор для начала вывода текста, а функция `End()` — указывает на окончание действий по выводу текста. Обе функции не имеют параметров и существовать друг без друга не могут. На каждый вызов `Begin()` обязан следовать вызов `End()`. А между ними происходит вызов функции `ID3DXFont::DrawText`:

```
pFont -> Begin();
pFont -> DrawText(StrokaTexta, -1, &Rec, DT_WORDBREAK, MyColor);
pFont -> End();
```

С помощью данной конструкции происходит прорисовка текста в заданной области вывода. Если вы хорошо знакомы с Win 32 API, то будьте внимательны: функция `ID3DXFont::DrawText` несколько отличается своими параметрами от `DrawText()` Win 32 API. Рассмотрим прототип функции `DrawText()`:

```
HRESULT DrawText(
    LPCSTR  pString,
    INT      Count,
    LPRECT   pRect,
    DWORD    Format,
    D3DCOLOR Color);
```

Функция `DrawText()` имеет следующие параметры:

- `pString` — указатель на строку текста. У нас это `StrokaTexta`;
- `Count` — это счетчик, определяющий количество символов в параметре `pString`. Чтобы не производить постоянный подсчет символов, достаточно поставить значение `-1`, и тогда подсчет будет производиться автоматически, что, признайте, весьма удобно;

- `pRect` — адрес того самого невидимого прямоугольника, координаты которого мы будем задавать значениями `x`, `y`, `x1` и `y1`;
- `Format` — с помощью этого значения происходит форматирование текста внутри области вывода. Само форматирование не влияет на ширину, высоту и другие параметры шрифта, а располагает выводимый текст относительно сторон прямоугольника. То есть двигает текст внутри области вывода вверх, вправо, влево и т. д. Все это достигается путем задания заранее определенных флагов. Рассмотрим некоторые из них:
 - `DT_BOTTOM` — текст выравнивается по нижней кромке области вывода;
 - `DT_CENTER` — текст выравнивается по центру области вывода;
 - `DT_RIGHT` — текст выравнивается по правому краю;
 - `DT_LEFT` — текст выравнивается по левому краю;
 - `DT_TOP` — текст выравнивается по верхней кромке;
 - `DT_END_ELLIPSIS` — если вы задали область вывода маленького размера, и ваш текст в нее не помещается, то весь не попавший текст будет заменен на троеточие;
 - `DT_WORDBREAK` — если текст не помещается в область вывода, он разбивается на несколько строк;
- `Color` — последний параметр, член структуры `D3DCOLOR`. Определяет цвет выводимого текста на экран с помощью задания значений цветовых режимов `XRb` или `ARGB`.

На этом разбор функции `DrawMyText()` закончен, и осталось произвести ее вызов с заданными параметрами в функции `RenderingDirect3D()`. Перед вызовом `EndScene()` запишем:

```
DrawMyText(pDirect3DDevice, hFont, "Ваш текст",  
           10, 10, 800, 800, D3DCOLOR_ARGB(250, 250, 250, 50));
```

Здесь все предельно просто: первый параметр — это устройство `Direct3D`, второй — созданный шрифт, третий параметр — строка текста, четыре последующих значения задают координаты области вывода текста и последний параметр — цвет текста (желтый). Таким образом происходит вывод текста в `Direct3D` на экран. Рассмотренный пример является самым простейшим вариантом, который можно только придумать, главное, чтобы вам был понятен весь смысл. Как упоминалось ранее, есть возможности создания класса, а также использования и других функций для вывода текста, определенных в классе `CD3DFont` библиотеки `DirectX 9`, например, `CD3DFont::DrawTextScaled`, `CD3DFont::Reder3DText`.

По окончании работы с указателем `pFont` не забываем освободить захваченные ресурсы:

```
if (pFont != NULL)
    pFont -> Release();
```

и поместить этот код в функцию `DeleteDirect3D()`.

Полноэкранный режим

С выводом текста на экран в `Direct3D` разобрались, теперь давайте перейдем из оконного режима в полноэкранный.

В функции `InitialDirect3D()`, в том месте, где происходит заполнение полей структуры `D3DPRESENT_PARAMETERS`, добавим еще четыре дополнительных параметра этой структуры:

```
Direct3DParametr.BackBufferWidth = GetSystemMetrics(SM_CXSCREEN);
Direct3DParametr.BackBufferHeight = GetSystemMetrics(SM_CYSCREEN);
Direct3DParametr.BackBufferCount = 3;
Direct3DParametr.FullScreen_RefreshRateInHz = Display.RefreshRate;
```

В первых двух строках с помощью функции `GetSystemMetrics()` мы берем размеры рабочего стола, т. е. разрешение экрана, и присваиваем полученный результат двум переменным `BackBufferWidth` и `BackBufferHeight`, отвечающим соответственно за ширину и высоту экрана. Функция Win 32 API `GetSystemMetrics()` принимает всего один параметр и имеет только два варианта вызова. С константой `SM_CXSCREEN`, определяющей размер рабочей поверхности экрана монитора по оси *X* (горизонталь) и `SM_CYSCREEN` — по оси *Y* (вертикаль).

Третье поле — `BackBufferCount` — устанавливает количество заданных буферов, равное трем, что несколько ускоряет и улучшает качество рендеринга.

И в последнем поле устанавливается частота обновления экрана. Строкой

```
Direct3DParametr.FullScreen_RefreshRateInHz = Display.RefreshRate;
```

мы берем частоту обновления, установленную в настройках адаптера.

Во всех добавленных параметрах можно установить числовые значения, например, $1024 \times 768 \times 85$ Гц, но этот вариант привязан конкретно к определенным аппаратным настройкам компьютера и не имеет большой гибкости в исполнении. Так же есть возможность воспользоваться значениями переменной `Display`, определенной как объект структуры `D3DDISPLAYMODE`, например:

```
Direct3DParametr.BackBufferWidth = Display.Width;
Direct3DParametr.BackBufferHeight = Display.Height;
```

И еще один шаг, который необходимо сделать для перехода в полноэкранный режим, — это установить значение `Windowed` в `FALSE`:

```
Direct3DParametr.Windowed = FALSE;
```

Поле `Windowed` как раз и отвечает за режимы отображения, где `TRUE` — оконный режим и `FALSE` — полноэкранный.

Откомпилировав и запустив рассматриваемый пример, вы получите полноценное полноэкранное приложение, из которого выйти, к сожалению, невозможно (не принимая во внимание комбинацию клавиш `<Ctrl>+<Alt>+<Del>`), кнопка закрытия окна в данный момент не доступна. Во избежание этого в функцию `MainWinProc()` в цикл обработки сообщений необходимо добавить код, осуществляющий выход из запущенного приложения, скажем, по нажатию клавиши `<Esc>`:

```
case WM_KEYDOWN:
{
    if (wparam == VK_ESCAPE)
        PostQuitMessage(0);
    return 0;
}
```

С помощью `WM_KEYDOWN` добавляются события по обработке нажатых клавиш с клавиатуры. Если `wparam` равен `VK_ESCAPE`, где `VK_ESCAPE` — виртуальный код для клавиши `<Esc>`, то осуществляется выход из приложения с помощью функции `PostQuitMessage()`.

Все, что было изучено на этом уроке, приведено в листинге 9.1.

Листинг 9.1. Файл `Text.cpp`

```
//-----
//  Text.cpp
//  Текст в Direct3D
//-----
#include <windows.h>    // подключаем заголовочный файл Windows
#include <d3d9.h>        // подключаем заголовочный файл DirectX 9 SDK
#include <d3dx9.h>       // подключаем заголовочный файл из D3DX
                        // для работы с матрицами
#include <mmsystem.h>    // подключаем системную библиотеку
#include <d3dx9core.h>   // подключаем системный заголовочный файл
//-----
//  глобальные переменные
```

```

//-----
LPDIRECT3D9 pDirect3D          = NULL;    // 3D-объект
LPDIRECT3DDEVICE9 pDirect3DDevice = NULL;  // устройство
LPDIRECT3DVERTEXBUFFER9 pBufferVershin = NULL; // буфер вершин
LPDIRECT3DINDEXBUFFER9 pBufferIndex   = NULL; // индексный буфер
LPD3DXFONT pFont                  = NULL;    // шрифт Direct3D

RECT Rec;                        // прямоугольник
HFONT hFont;                    // шрифт

struct CUSTOMVERTEX
{
    FLOAT X, Y, Z;              // координаты вершины
    FLOAT nx, ny, nz;          // нормали
};

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_NORMAL) // формат вершины
//-----
// функция
// IDirect3D()
// инициализация Direct3D
//-----
HRESULT InitialDirect3D(HWND hwnd)
{
    if (NULL == (pDirect3D = Direct3DCreate9(D3D_SDK_VERSION)))
        return E_FAIL;
    D3DDISPLAYMODE Display;
    if (FAILED(pDirect3D -> GetAdapterDisplayMode(D3DADAPTER_DEFAULT,
                                                    &Display)))
        return E_FAIL;
    D3DPRESENT_PARAMETERS Direct3DParametr;
    ZeroMemory(&Direct3DParametr, sizeof(Direct3DParametr));
    Direct3DParametr.Windowed = TRUE;
    Direct3DParametr.SwapEffect = D3DSWAPEFFECT_DISCARD;
    Direct3DParametr.BackBufferFormat = Display.Format;
    Direct3DParametr.EnableAutoDepthStencil = TRUE;
    Direct3DParametr.AutoDepthStencilFormat = D3DFMT_D16;
    Direct3DParametr.BackBufferWidth = GetSystemMetrics(SM_CXSCREEN);
    Direct3DParametr.BackBufferHeight = GetSystemMetrics(SM_CYSCREEN);
    Direct3DParametr.BackBufferCount = 3;
    Direct3DParametr.FullScreen_RefreshRateInHz = Display.RefreshRate;
}

```

```

    if (FAILED(pDirect3D -> CreateDevice(D3DADAPTER_DEFAULT,
        D3DDEVTYPE_HAL, hwnd,
        D3DCREATE_SOFTWARE_VERTEXPROCESSING,
        &Direct3DParametr, &pDirect3DDevice)))
    return E_FAIL;
//    включаем отсечение в Direct3D
pDirect3DDevice -> SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);
//    подключаем Z-буфер
pDirect3DDevice -> SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
return S_OK;
}
//-----
//    функция
//    InitialObject()
//    инициализация вершин
//-----
HRESULT InitialObject()
{
    CUSTOMVERTEX Vershin[] =
    {
        {1.0f, -1.0f, -1.0f, 0.0f, 0.0f, -1.0f},    // A
        {1.0f, 1.0f, -1.0f, 0.0f, 0.0f, -1.0f},    // B
        {-1.0f, 1.0f, -1.0f, 0.0f, 0.0f, -1.0f},   // C
        {-1.0f, -1.0f, -1.0f, 0.0f, 0.0f, -1.0f},  // D

        {-1.0f, -1.0f, -1.0f, -1.0f, 0.0f, 0.0f}, // A2
        {-1.0f, 1.0f, -1.0f, -1.0f, 0.0f, 0.0f},  // B2
        {-1.0f, 1.0f, 1.0f, -1.0f, 0.0f, 0.0f},   // C2
        {-1.0f, -1.0f, 1.0f, -1.0f, 0.0f, 0.0f},  // D2

        {-1.0f, -1.0f, 1.0f, 0.0f, 0.0f, 1.0f},   // A3
        {-1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f},    // B3
        {1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f},      // C3
        {1.0f, -1.0f, 1.0f, 0.0f, 0.0f, 1.0f},     // D3

        {1.0f, -1.0f, 1.0f, 1.0f, 0.0f, 0.0f},    // A4
        {1.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f},     // B4
        {1.0f, 1.0f, -1.0f, 1.0f, 0.0f, 0.0f},    // C4
        {1.0f, -1.0f, -1.0f, 1.0f, 0.0f, 0.0f},   // D4
    }
}

```

```

{1.0f, -1.0f, -1.0f, 0.0f, -1.0f, 0.0f}, // A5
{-1.0f, -1.0f, -1.0f, 0.0f, -1.0f, 0.0f}, // B5
{-1.0f, -1.0f, 1.0f, 0.0f, -1.0f, 0.0f}, // C5
{1.0f, -1.0f, 1.0f, 0.0f, -1.0f, 0.0f}, // D5

{1.0f, 1.0f, 1.0f, 0.0f, 1.0f, 0.0f}, // A6
{-1.0f, 1.0f, 1.0f, 0.0f, 1.0f, 0.0f}, // B6
{-1.0f, 1.0f, -1.0f, 0.0f, 1.0f, 0.0f}, // C6
{1.0f, 1.0f, -1.0f, 0.0f, 1.0f, 0.0f}, // D6

};

const unsigned short Index[]={
0,1,2,      2,3,0,
4,5,6,      6,7,4,
8,9,10,     10,11,8,
12,13,14,   14,15,12,
16,17,18,   18,19,16,
20,21,22,   22,23,20,
};

// создаем буфер вершин
if (FAILED(pDirect3DDevice -> CreateVertexBuffer(
                                36*sizeof(CUSTOMVERTEX),
                                0, D3DFVF_CUSTOMVERTEX,
                                D3DPOOL_DEFAULT, &pBufferVershin, NULL)))

return E_FAIL;

// блокируем
VOID* pBV;
if (FAILED(pBufferVershin -> Lock(0,sizeof(Vershin), (void**)&pBV,0)))
return E_FAIL;

// копируем
memcpy(pBV, Vershin, sizeof(Vershin));

// разблокируем
pBufferVershin -> Unlock();

// создаем индексный буфер
pDirect3DDevice -> CreateIndexBuffer(36*sizeof(Index), 0,
                                D3DFMT_INDEX16, D3DPOOL_DEFAULT,
                                &pBufferIndex, NULL);

// блокируем
VOID* pBI;

```

```

pBufferIndex -> Lock(0, sizeof(Index), (void**)&pBI, 0);
// копируем
memcpy(pBI, Index, sizeof(Index));
// разблокируем
pBufferIndex -> Unlock();
return S_OK;
}

//-----
// функция
// Matrix()
// мировая матрица, матрица вида, матрица проекции
//-----
VOID Matrix()
{
    D3DXMATRIX MatrixWorld, MatrixWorldX, MatrixWorldY; // мировая матрица
    D3DXMATRIX MatrixView; // матрица вида
    D3DXMATRIX MatrixProjection; // матрица проекции
    // MatrixWorld
    UINT Time = timeGetTime() % 5000;
    FLOAT Angle = Time * (2.0f * D3DX_PI) / 5000.0f;
    D3DXMatrixRotationX(&MatrixWorldX, Angle);
    D3DXMatrixRotationY(&MatrixWorldY, Angle);
    D3DXMatrixMultiply(&MatrixWorld, &MatrixWorldX, &MatrixWorldY);
    pDirect3DDevice -> SetTransform(D3DTS_WORLD, &MatrixWorld);
    // MatrixView
    D3DXMatrixLookAtLH(&MatrixView, &D3DXVECTOR3(0.0f, 0.0f, -11.0f),
                      &D3DXVECTOR3(0.0f, 0.0f, 0.0f),
                      &D3DXVECTOR3(0.0f, 1.0f, 0.0f));
    pDirect3DDevice -> SetTransform(D3DTS_VIEW, &MatrixView);
    // MatrixProjection
    D3DXMatrixPerspectiveFovLH(&MatrixProjection, D3DX_PI/4,
                              1.0f, 1.0f, 100.0f);
    pDirect3DDevice -> SetTransform(D3DTS_PROJECTION, &MatrixProjection);
}

//-----
// функция
// LightMaterial()
// инициализация света и материала

```

```

//-----
VOID LightMaterial()
{
    D3DMATERIAL9 Material;    // материал
    D3DLIGHT9 Light;          // цвет
    // установим материал
    ZeroMemory( &Material, sizeof(D3DMATERIAL9) );
    Material.Diffuse.r = Material.Ambient.r = 1.0f;
    Material.Diffuse.g = Material.Ambient.g = 1.0f;
    Material.Diffuse.b = Material.Ambient.b = 0.0f;
    Material.Diffuse.a = Material.Ambient.a = 1.0f;
    pDirect3DDevice -> SetMaterial(&Material);
    D3DXVECTOR3 VectorDir;
    // установим свет
    ZeroMemory( &Light, sizeof(D3DLIGHT9) );
    Light.Type = D3DLIGHT_DIRECTIONAL;
    Light.Diffuse.r = 1.0f;
    Light.Diffuse.g = 1.0f;
    Light.Diffuse.b = 1.0f;
    Light.Range = 1000.0f;
    // установим нормаль
    VectorDir = D3DXVECTOR3(0.0f, 0.0f, 1.0f),
    D3DXVec3Normalize( (D3DXVECTOR3*)&Light.Direction, &VectorDir);

    pDirect3DDevice -> SetLight(0, &Light);
    pDirect3DDevice -> LightEnable(0, TRUE);

    pDirect3DDevice -> SetRenderState(D3DRS_LIGHTING, TRUE);
    pDirect3DDevice -> SetRenderState(D3DRS_AMBIENT, 0);
}
//-----
// функция
// DrawMyText()
// вывод текста на экран
//-----
VOID DrawMyText(LPDIRECT3DDEVICE9 pDirect3DDevice, HFONT hFont,
                char* StrokaTexta, int x, int y, int x1,
                int y1, D3DCOLOR MyColor)

```

```

{
    // создаем шрифт
    hFont = CreateFont(30, 10, 0, 0, FW_NORMAL, FALSE, FALSE, 0, 1, 0,
                      0, 0, DEFAULT_PITCH|FF_MODERN, "Arial");

    // координаты прямоугольника
    Rec.left   = x;
    Rec.top    = y;
    Rec.right  = x1;
    Rec.bottom = y1;

    // инициализируем шрифт
    D3DXCreateFont(pDirect3DDevice, hFont, &pFont);
    // начало
    pFont -> Begin();
    // вывод текста
    pFont -> DrawText(StrokaTexta, -1, &Rec, DT_WORDBREAK, MyColor);
    // Конец
    pFont -> End();
}

//-----
// функция
// RenderingDirect3D()
// рисование
//-----

VOID RenderingDirect3D()
{
    if (pDirect3DDevice == NULL)    // проверяем ошибки
        return;

    pDirect3DDevice -> Clear(0, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
                           D3DCOLOR_XRGB(60, 100, 150), 1.0f, 0);

    // начало сцены
    pDirect3DDevice -> BeginScene();
    // здесь происходит прорисовка сцены
    LightMaterial();
    Matrix();

    pDirect3DDevice -> SetStreamSource(0, pBufferVershin, 0,
                                       sizeof(CUSTOMVERTEX));

    pDirect3DDevice -> SetFVF(D3DFVF_CUSTOMVERTEX);
    pDirect3DDevice -> SetIndices(pBufferIndex);
}

```

```

// вывод объекта
pDirect3DDevice -> DrawIndexedPrimitive(D3DPT_TRIANGLELIST,
                                         0, 0, 36, 0, 12);

// вывод текста
DrawMyText(pDirect3DDevice, hFont, "Текст в DirectX",
           10, 10, 500, 700, D3DCOLOR_ARGB(250, 250, 250, 50))

// конец сцены
pDirect3DDevice -> EndScene();

// представляем на экране
pDirect3DDevice -> Present(NULL, NULL, NULL, NULL);
}

//-----
// функция
// DeleteDirect3D()
// освобождает захваченные ресурсы
//-----
VOID DeleteDirect3D()
{
    if (pFont != NULL)
        pFont -> Release();

    if (pBufferIndex != NULL)
        pBufferIndex -> Release();

    if (pBufferVershin != NULL)
        pBufferVershin -> Release();

    if (pDirect3DDevice != NULL)
        pDirect3DDevice -> Release();

    if (pDirect3D != NULL)
        pDirect3D -> Release();
}

//-----
// функция
// MainWinProc()
// здесь происходит обработка сообщений

```

```
//-----  
  
LRESULT CALLBACK MainWinProc(HWND    hwnd,  
                               UINT    msg,  
                               WPARAM  wparam,  
                               LPARAM  lparam)  
{  
    switch(msg)  
    {  
        case WM_DESTROY:  
        {  
            DeleteDirect3D();  
            PostQuitMessage(0);  
            return(0);  
        }  
        case WM_KEYDOWN:  
        {  
            if (wparam == VK_ESCAPE)  
                PostQuitMessage(0);  
            return 0;  
        }  
    }  
    return DefWindowProc(hwnd, msg, wparam, lparam);  
}  
  
//-----  
//  функция  
//  WinMain  
//  входная точка приложения  
//-----  
  
int WINAPI WinMain(HINSTANCE  hinstance,  
                  HINSTANCE  hpreinstance,  
                  LPSTR      lpcmdline,  
                  int         ncmdshow)  
{  
    WINDCLASSEX windowclass;    // создаем класс  
    HWND hwnd;                  // создаем дескриптор окна  
    MSG msg;                     // идентификатор сообщения
```

```

// определим класс окна WINDCLASSEX
windowclass.cbSize = sizeof(WINDCLASSEX);
windowclass.style = CS_DBLCLKS|CS_OWNDC|CS_HREDRAW|CS_VREDRAW;
windowclass.lpfnWndProc = MainWinProc;
windowclass.cbClsExtra = 0;
windowclass.cbWndExtra = 0;
windowclass.hInstance = hinstance;
windowclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
windowclass.hCursor = LoadCursor(NULL, IDC_ARROW);
windowclass.hbrBackground = (HBRUSH)GetStockObject(GRAY_BRUSH);
windowclass.lpszMenuName = NULL;
windowclass.lpszClassName = "WINDOWSCCLASS";
windowclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
// регистрируем класс
if (!RegisterClassEx(&windowclass))
return(0);
// теперь, когда класс зарегистрирован, можно создать окно
if (! (hwnd = CreateWindowEx(NULL, "WINDOWSCCLASS", "Текст в Direct3D",
                               WS_OVERLAPPEDWINDOW|WS_VISIBLE, 0, 0,
                               GetSystemMetrics(SM_CXSCREEN),
                               GetSystemMetrics(SM_CYSCREEN),
                               NULL,NULL,hinstance, NULL)))

return 0;
if (SUCCEEDED(InitialDirect3D(hwnd)))
{
    if (SUCCEEDED(InitialObject() ))
    {
        ShowWindow(hwnd, SW_SHOWDEFAULT);
        UpdateWindow(hwnd);
        ZeroMemory(&msg, sizeof(msg));
        while (msg.message != WM_QUIT)
        {
            if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
            {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
            }
            else

```

```
        RenderingDirect3D();  
    }  
}  
}  
return 0;  
}
```

Итоги урока

На этом уроке был рассмотрен самый простейший способ вывода текста на экран. Мы создали функцию, с помощью которой можем осуществлять вывод текста на экран в любом необходимом месте, с любыми нужными для нас свойствами — цветом, размером, стилем, форматированием и т. д. Также был осуществлен переход в полноэкранный режим работы. На *уроке 10* мы изучим текстурирование, т. е. наложение простой растровой картинки на объект.

Урок 10



Текстурирование

Текстура — это растровая картинка, наложенная на объект. Очень хороший пример: ваши домашние обои, наклеенные на голые и некрасивые стены. Сама текстура, это ни что иное, как плоское двумерное изображение, созданное в любом графическом редакторе. Все текстуры обычно хранятся в файлах с расширениями `bmp`, `gif`, `jpeg`, `tiff` в каталоге вашего приложения, и загружаются с помощью определенных в `Direct3D` функций. Когда вы накладываете текстуру на объект, то способ такого подбора поверхности называется *текстурированием*. Работа с текстурами открывает большие возможности в программировании компьютерных игр. Например, нарисовав любой прямоугольник и наложив на него текстуру в виде блока кирпичей, вы получите кирпичную стену, а наложив на все тот же прямоугольник другую текстуру, оформленную в виде блока деревянных досок, получите деревянную стену. Сфера применения текстур вполне очевидна.

Чтобы наложить текстуру на примитив, необходимо воспользоваться библиотечной функцией `Direct3D` и указать текстурные координаты. Эти координаты никаким образом не зависят от фактического размера объекта и определяются в своей небольшой системе координат. Посмотрим на рис. 10.1.

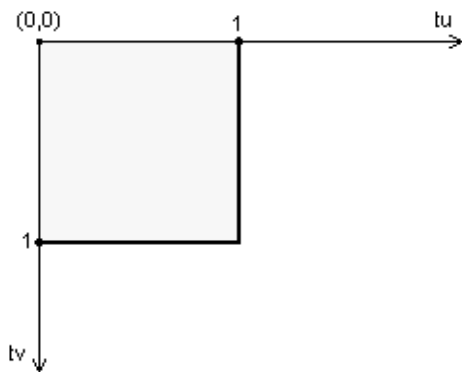


Рис. 10.1. Текстурная система координат

Верхний левый угол — это начало текстурных координат. Горизонтальная ось tu и вертикальная — tv , а вся система координат применяется также, как при работе с двумерными объектами. Однако координаты самой текстуры определены двумя значениями в диапазоне от 0 до 1. То есть начало текстуры — 0, а конец — это 1, вне зависимости от размера самой текстуры и размера объекта, на который накладывается эта текстура. Но есть возможность использовать и другие значения. Скажем, если вы вместо 1 установите 2 или более, то текстура размножится на всем примитиве, а если поставить значение, например, 0.5, то текстура, наоборот, растянется. Эта операция называется "тайлинг" (tiling), говорят, что вы накладываете текстуру и "тайлите" ее. Это очень распространенный метод. Вернемся к той же кирпичной стене, представьте ее размером во весь экран. Чтобы сделать из нее действительно кирпичную стену, необходима небольшая текстура из десятка кирпичей, которую вы просто размножаете, вместо того чтобы рисовать всю стену в полный размер.

Что касается размера самой текстуры, то лучше когда ее стороны кратны двум: 2, 4, 16 и т. д. В любом случае текстуры рисует специальный текстурный художник и вас это не должно особо касаться, но если же вы все-таки делаете их сами, то возьмите за правило делать стороны кратными двум.

Теперь перейдем к реализации программного кода нашего урока. Первое, что мы должны сделать, это изменить структуру `CUSTOMVERTEX`, добавив координаты для определения текстур:

```
struct CUSTOMVERTEX
{
    FLOAT X, Y, Z;
    FLOAT nx, ny, nz;
    FLOAT tu, tv;
};
```

Соответственно в описании формата вершин добавится текстурный формат:

```
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_NORMAL|D3DFVF_TEX1)
```

Загрузка текстуры

Создадим новую переменную для представления нашей текстуры. Для этих целей в Direct3D определен специальный интерфейс `IDirect3DTexture9`:

```
LPDIRECT3DTEXTURE9 Textura001;
```

Так мы создали указатель на интерфейс, в котором будет храниться вся информация о загружаемой текстуре.

Далее в функцию `InitialObject()` добавим строку, с помощью которой произойдет загрузка текстуры из каталога нашего приложения. В этой стро-

ке будет использоваться функция `D3DXCreateTextureFromFile()`, название которой говорит само за себя. Рассмотрим ее прототип:

```
HRESULT D3DXCreateTextureFromFile(  
    LPDIRECT3DDEVICE9    pDevice,  
    LPCTSTR               pSrcFile,  
    LPDIRECT3DTEXTURE9*  ppTexture);
```

Функция `D3DXCreateTextureFromFile()` имеет следующие параметры:

- `pDevice` — указатель на интерфейс `IDirect3DDevice9`, показывающий, к какому устройству присоединяется текстура. В нашем случае это устройство `pDirect3DDevice`;
- `pSrcFile` — имя файла загружаемой текстуры;
- `ppTexture` — адрес указателя на интерфейс `IDirect3DTexture9`, являющийся результатом всей операции.

Перед тем как создать буфер вершин в функции `InitialObject()`, запишем:

```
if (FAILED(D3DXCreateTextureFromFile(pDirect3DDevice,  
                                     Directx.jpg", &pTextura001)));  
return E_FAIL;
```

В качестве загружаемой текстуры послужит незамысловатая текстура, которую я сделал и назвал `Directx.jpg` и положил в каталог `Urok10` рассматриваемого проекта (см. компакт-диск).

Загрузив текстуру в память, необходимо отразить ее в сцене, но сначала мы должны указать текстурные координаты, т. е. где и каким образом предполагаемая текстура будет накладываться на объект. Для этого в структуре `Vershin` добавляются две текстурные координаты `tu` и `tv`. На их значениях, определяющих наложение текстуры на примитив, давайте остановимся подробнее.

Первая лицевая сторона нашего куба состоит из двух треугольников и четырех вершин. Первая вершина располагается в точке `A`, вторая в точке `B`, третья и четвертая соответственно в точках `C` и `D`. Мы знаем, что текстурные координаты никакого отношения не имеют к фактическому размеру объекта и задаются в своей системе координат, начальная точка отсчета которой находится в верхнем левом углу. Там, где у нас третья вершина, точка `C`. Для наглядности посмотрите на рис. 10.2.

На рисунке очень хорошо видно, как определяются координаты для текстуры. Точка `C` — это третья вершина квадрата и имеет значение $(0,0)$, точка `B` находится на горизонтальной оси `tu`, ее значение $(0,1)$, точка `D` — $(1,0)$ и последняя точка `A` — $(1,1)$. В скобках значения записываются сначала для оси `tu`, а после запятой — для оси `tv`. Это пример лицевой стороны куба. Все остальные координаты вершин записываются точно так же. При этом

необходимо учитывать, что порядок расположения вершин верхней и нижней сторон куба несколько другой, и если загружаемая вами текстура имеет направленный рисунок, то об этом надо помнить.

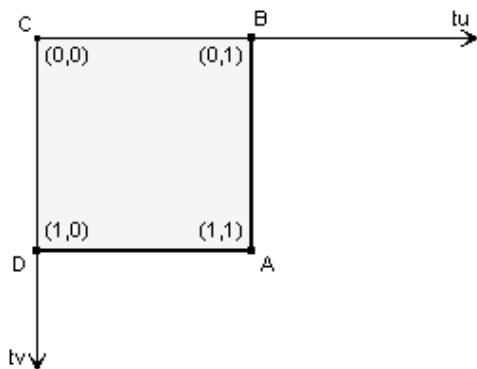


Рис. 10.2. Наложение текстуры на лицевую сторону куба

Посмотрим, как будет выглядеть структура `Vertexin` после добавления в нее текстурных координат:

```
CUSTOMVERTEX Vertexin[] =
{
    // X      Y      Z      nx      ny      nz      tu      tv
    {1.0f, -1.0f, -1.0f, 0.0f, 0.0f, -1.0f, 1.0f, 1.0f},    // A
    {1.0f, 1.0f, -1.0f, 0.0f, 0.0f, -1.0f, 0.0f, 1.0f},   // B
    {-1.0f, 1.0f, -1.0f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f},  // C
    {-1.0f, -1.0f, -1.0f, 0.0f, 0.0f, -1.0f, 1.0f, 0.0f}, // D
}
```

В этом примере текстурные значения, заданные для вершин, были от 0 до 1, текстура наклеивалась один в один на стороны куба. Когда закончим этот урок, попробуйте задать значения больше и меньше 1.

Рендеринг текстурированного объекта

Следующая наша задача — отобразить всю сцену на экране монитора. Перейдем в функцию `RenderingDirect3D()`, в место начала сцены от функции `BeginScene()` и рассмотрим такую запись:

```
pDirect3DDevice -> BeginScene();
LightMaterial();
Matrix();
```

```
pDirect3DDevice -> SetStreamSource(0, pBufferVerшин, 0,
                                   sizeof(CUSTOMVERTEX));

pDirect3DDevice -> SetFVF(D3DFVF_CUSTOMVERTEX);

pDirect3DDevice -> SetIndices(pBufferIndex);

pDirect3DDevice -> SetTexture(0, pTextura001);

pDirect3DDevice -> SetTextureStageState(0, D3DTSS_COLORARG1,
                                         D3DTA_TEXTURE);

pDirect3DDevice -> SetTextureStageState(0, D3DTSS_COLOROP,
                                         D3DTOP_MODULATE);

pDirect3DDevice -> DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0,
                                         0, 36, 0, 12);

pDirect3DDevice -> EndScene();
```

Первая вызываемая функция это `IDirect3DDevice9::SetTexture`. Рассмотрим ее прототип:

```
HRESULT SetTexture(
    DWORD Stage,
    IDirect3DBaseTexture9* pTexture);
```

Функция `SetTexture()` имеет следующие параметры:

- `Stage` — это идентификатор, определяемый для каждой текстуры. Один объект может иметь до восьми текстур, поэтому каждая текстура должна иметь свой идентификатор. Отсчет начинается с 0, заканчивается 7;
- `pTexture` — указатель на интерфейс `IDirect3DBaseTexture9`, где и находится сама загружаемая текстура.

С помощью этой функции устанавливается идентификатор для каждой текстуры, а значит, увеличивается счетчик, как уже говорилось, до 7, поэтому, когда текстура не нужна, установите ее значение в `NULL`. Тем самым вы предотвратите утечку памяти по завершении работы приложения.

Далее следуют вызовы функции `IDirect3DDevice9::SetTextureStageState`, с помощью которой устанавливаются состояния и текстурные операции. Здесь необходимо сосредоточиться, потому что последует трудный материал с вложенными друг в друга структурами. Рассмотрим прототип функции `SetTextureStageState()`:

```
HRESULT SetTextureStageState(
    DWORD Stage,
    D3DTEXTURESTAGESTATETYPE Type,
    D3DPRD Value);
```

Функция `SetTextureStageState()` имеет следующие параметры:

- ❑ `Stage` — идентификатор, определенный для текстуры в границах от 0 до 7 (еще говорят — текстурный уровень);
- ❑ `Type` — параметр из перечисляемого типа `D3DTEXTURESTAGESTATETYPE`, устанавливающий состояние для загружаемой текстуры;
- ❑ `Value` — эта переменная определяет текстурную операцию для состояния текстуры, установленную в параметре `Type`.

Действие последнего блока из двух параметров аналогично разобранный на уроке 6 функции `SetRenderState()`, где последующий параметр влияет на состояние предыдущего.

Параметр `Type` — член перечисляемого типа и имеет очень много значений. Давайте посмотрим на них, но разберем только те, которые используем:

```
typedef enum D3DTEXTURESTAGESTATETYPE (
    D3DTSS_COLOROP                = 1,
    D3DTSS_COLORARG1              = 2,
    D3DTSS_COLORARG2              = 3,
    D3DTSS_ALPHAOP                = 4,
    D3DTSS_ALPHAARG1              = 5,
    D3DTSS_ALPHAARG2              = 6,
    D3DTSS_BUMPENVMAT00           = 7,
    D3DTSS_BUMPENVMAT01           = 8,
    D3DTSS_BUMPENVMAT10           = 9,
    D3DTSS_BUMPENVMAT11           = 10,
    D3DTSS_TEXCOORDINDEX          = 11,
    D3DTSS_BUMPENVLSCALE          = 22,
    D3DTSS_BUMPENVLOFFSET        = 23,
    D3DTSS_TEXTURETRANSFORMFLAGS = 24,
    D3DTSS_COLORARG0              = 26,
    D3DTSS_ALPHAARG0              = 27,
    D3DTSS_RESULTARG              = 28,
    D3DTSS_CONSTANT               = 32,
    D3DTSS_FORCE_DWORD            = 0x7fffffff
} D3DTEXTURESTAGESTATETYPE;
```

Все эти значения определяют текстурное состояние для загружаемой текстуры и работают так, как определено в переменной `Value`. Сама же переменная `Value` определяется аргументными флагами или флагами модификации. Флаги модификации устанавливают состояние текстур и принимают значения из перечисляемого типа `D3DTEXTUREOP`.

Рассмотрим этот тип:

```
typedef enum _D3DTEXTUREOP {
    D3DTOP_DISABLE                = 1,
    D3DTOP_SELECTARG1             = 2,
    D3DTOP_SELECTARG2            = 3,
    D3DTOP_MODULATE               = 4,
    D3DTOP_MODULATE2X             = 5,
    D3DTOP_MODULATE4X            = 6,
    D3DTOP_ADD                    = 7,
    D3DTOP_ADDSIGNED              = 8,
    D3DTOP_ADDSIGNED2X           = 9,
    D3DTOP_SUBTRACT               = 10,
    D3DTOP_ADDSMOOTH              = 11,
    D3DTOP_BLENDDIFFUSEALPHA      = 12,
    D3DTOP_BLENDTEXTUREALPHA     = 13,
    D3DTOP_BLENDFACTORALPHA      = 14,
    D3DTOP_BLENDTEXTUREALPHAPM   = 15,
    D3DTOP_BLENDCURRENTALPHA     = 16,
    D3DTOP_PREMODULATE           = 17,
    D3DTOP_MODULATEALPHA_ADDCOLOR = 18,
    D3DTOP_MODULATECOLOR_ADDALPHA = 19,
    D3DTOP_MODULATEINVALPHA_ADDCOLOR = 20,
    D3DTOP_MODULATEINVCOLOR_ADDALPHA = 21,
    D3DTOP_BUMPENVMAP            = 22,
    D3DTOP_BUMPENVMAPLUMINANCE   = 23,
    D3DTOP_DOTPRODUCT3           = 24,
    D3DTOP_MULTIPLYADD           = 25,
    D3DTOP_LERP                  = 26,
    D3DTOP_FORCE_DWORD          = 0x7fffffff
} D3DTEXTUREOP;
```

Итак, подведем итог. В параметре `Type` устанавливается текстурное состояние, а в параметре `Value` определяется, как будет работать значение в параметре `Type`.

Теперь перейдем непосредственно к коду. В первом вызове функции `SetTextureStageState()` в параметре `Type` использовалось значение `D3DTSS_COLORARG1`.

Рассмотрим это значение, а также флаги для параметра `Value`:

- ❑ `D3DTSS_COLORARG1 (2)` — определяет первый и второй цветовой аргументы для состояния текстуры. Принимает значения, определенные в переменной `Value`, которые могут быть такими:
 - `D3DTA_TEXTURE` — это значение по умолчанию является аргументным флагом и определяет цвет состояния текстуры;
 - `D3DTA_DIFFUSE` — это цвет вершин с применением затемнения по Гуро. В том случае, когда вершины не определены никакой цветовой компонентой, устанавливается белый цвет (`0xffffffff`);
 - `D3DTA_CURRENT` — аргументный флаг. Он устанавливает предыдущее текстурное состояние. Если идентификатор `Stage` установлен в 0 (своего рода первый текстурный уровень), то значение будет одинаковым с `D3DTA_DIFFUSE`.

Второй вызов функции `SetTextureStageState()` для параметра `Type` был со значением `D3DTSS_COLOROP`:

- ❑ `D3DTSS_COLOROP` — определяет, как будет происходить смешение цвета текстур и может принимать значения, определенные в переменной `Value` из перечисляемого типа `D3DTEXTUREOP`. Рассмотрим эти значения:
 - `D3DTOP_SELECTARG1 (2)` — состояние текстуры. Для вывода может использоваться первый или второй альфа-аргумент (цветовой);
 - `D3DTOP_MODULATE` — с помощью этого флага происходит умножение `ARG1` и `ARG2`. При использовании этого флага, в отличие от первого, освещение в сцене будет включено;
 - `D3DTOP_MODULATE2X (4x)` — перемножает `ARG1` и `ARG2`, и вдобавок сдвигает результат на 1 или 2 бита, что приводит к более яркому изображению;
 - `D3DTOP_ADD` — складывает `ARG1` и `ARG2`;
 - `D3DTOP_SUBTRACT` — вычитает `ARG2` из `ARG1`;
 - `D3DTOP_DISABLE` — полностью отключает вывод текстур для указанного значения в идентификаторе `Stage`.

После того как поработали с текстурами, не забываем убрать за собой "мусор", записав такие строки кода в функции `DeleteDirect3D()`:

```
if (pTextura001 != NULL )  
pTextura001 -> Release();
```

Весь итоговый код приведен в листинге 10.1.

Листинг 10.1. Файл Textura.cpp

```

//-----
//  Textura.cpp
//  используем текстуру
//-----
#include <windows.h>      // подключаем заголовочный файл Windows
#include <d3d9.h>          // подключаем заголовочный файл DirectX 9 SDK
#include <d3dx9.h>         // подключаем заголовочный файл из D3DX
                          // для работы с матрицами
#include <mmsystem.h>      // подключаем системную библиотеку
#include <d3dx9core.h>     // подключаем системный заголовочный файл
//-----
//  глобальные переменные
//-----
LPDIRECT3D9 pDirect3D      = NULL;      // 3D-объект
LPDIRECT3DDEVICE9 pDirect3DDevice = NULL; // устройство
LPDIRECT3DVERTEXBUFFER9 pBufferVershin = NULL; // буфер вершин
LPDIRECT3DINDEXBUFFER9 pBufferIndex   = NULL; // индексный буфер
LPDIRECT3DTEXTURE9 pTextura001        = NULL; // текстура
LPD3DXFONT pFont                  = NULL; // шрифт Diect3D
RECT Rec;                          // прямоугольник
HFONT hFont;                       // шрифт
struct CUSTOMVERTEX
{
    FLOAT X, Y, Z;      // координаты вершины
    FLOAT nx, ny, nz;   // нормали
    FLOAT tu, tv;       // текстура
};
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1)
//-----
//  функция
//  InitialDirect3D()
//  инициализация Direct3D
//-----
HRESULT InitialDirect3D(HWND hwnd)
{
    if (NULL == (pDirect3D = Direct3DCreate9(D3D_SDK_VERSION)))

```

```

return E_FAIL;
D3DDISPLAYMODE Display;
if (FAILED(pDirect3D -> GetAdapterDisplayMode(D3DADAPTER_DEFAULT,
                                              &Display)))

return E_FAIL;
D3DPRESENT_PARAMETERS Direct3DParametr;
ZeroMemory(&Direct3DParametr, sizeof(Direct3DParametr));
Direct3DParametr.Windowed = FALSE;
Direct3DParametr.SwapEffect = D3DSWAPEFFECT_DISCARD;
Direct3DParametr.BackBufferFormat = Display.Format;
Direct3DParametr.EnableAutoDepthStencil = TRUE;
Direct3DParametr.AutoDepthStencilFormat = D3DFMT_D16;
Direct3DParametr.BackBufferWidth = GetSystemMetrics(SM_CXSCREEN);;
Direct3DParametr.BackBufferHeight = GetSystemMetrics(SM_CYSCREEN);;
Direct3DParametr.BackBufferCount = 3;
Direct3DParametr.FullScreen_RefreshRateInHz = Display.RefreshRate;
if (FAILED(pDirect3D -> CreateDevice(D3DADAPTER_DEFAULT,
                                     D3DDEVTYPE_HAL, hwnd,
                                     D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                     &Direct3DParametr, &pDirect3DDevice)))

return E_FAIL;
// включаем отсечение Direct3D
pDirect3DDevice -> SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);
// подключаем Z-буфер
pDirect3DDevice -> SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
return S_OK;
}
//-----
// функция
// InitialObject()
// инициализация вершин
//-----
HRESULT InitialObject()
{
    CUSTOMVERTEX Vershin[] =
    {
        // X      Y      Z      nx      ny      nz      tu      tv
        {1.0f, -1.0f, -1.0f, 0.0f, 0.0f, -1.0f, 1.0f, 1.0f},    // A
        {1.0f, 1.0f, -1.0f, 0.0f, 0.0f, -1.0f, 0.0f, 1.0f},    // B
    }

```

[illegible]

```

return E_FAIL;
// создаем буфер вершин
if (FAILED(pDirect3DDevice -> CreateVertexBuffer(
    36*sizeof(CUSTOMVERTEX),
    0, D3DFVF_CUSTOMVERTEX, D3DPOOL_DEFAULT,
    &pBufferVershin, NULL)))

return E_FAIL;
// блокируем
VOID* pBV;
if (FAILED(pBufferVershin -> Lock(0, sizeof(Vershin), (void*)&pBV, 0)))
return E_FAIL;
// копируем
memcpy(pBV, Vershin, sizeof(Vershin));
// разблокируем
pBufferVershin -> Unlock();
// создаем индексный буфер
pDirect3DDevice -> CreateIndexBuffer(36*sizeof(Index),
    0, D3DFMT_INDEX16, D3DPOOL_DEFAULT,
    &pBufferIndex, NULL);

// блокируем
VOID* pBI;
pBufferIndex -> Lock(0, sizeof(Index), (void*)&pBI, 0);
// копируем
memcpy(pBI, Index, sizeof(Index));
// разблокируем
pBufferIndex -> Unlock();
return S_OK;
}

//-----
// функция
// Matrix()
// мировая матрица, матрица вида, матрица проекции
//-----
VOID Matrix()
{
    D3DXMATRIX MatrixWorld, MatrixWorldX, MatrixWorldY; // мировая матрица
    D3DXMATRIX MatrixView; // матрица вида
    D3DXMATRIX MatrixProjection; // матрица проекции
    // MatrixWorld

```

```

UINT Time = timeGetTime() % 5000;
FLOAT Angle = Time * (2.0f * D3DX_PI) / 5000.0f;
D3DXMatrixRotationX(&MatrixWorldX, Angle);
D3DXMatrixRotationY(&MatrixWorldY, Angle);
D3DXMatrixMultiply(&MatrixWorld, &MatrixWorldX, &MatrixWorldY);
pDirect3DDevice -> SetTransform(D3DTS_WORLD, &MatrixWorld);
// MatrixView
D3DXMatrixLookAtLH(&MatrixView, &D3DXVECTOR3(0.0f, 0.0f, -11.0f),
                  &D3DXVECTOR3(0.0f, 0.0f, 0.0f),
                  &D3DXVECTOR3(0.0f, 1.0f, 0.0f));
pDirect3DDevice -> SetTransform(D3DTS_VIEW, &MatrixView);
// MatrixProjection
D3DXMatrixPerspectiveFovLH(&MatrixProjection, D3DX_PI/4,
                          1.0f, 1.0f, 100.0f);
pDirect3DDevice -> SetTransform(D3DTS_PROJECTION, &MatrixProjection);
}
//-----
// функция
// LightMaterial()
// инициализация света и материала
//-----
VOID LightMaterial()
{
    D3DMATERIAL9 Material;    // материал
    D3DLIGHT9 Light;          // свет
    // установим материал
    ZeroMemory(&Material, sizeof(D3DMATERIAL9));
    Material.Diffuse.r = Material.Ambient.r = 1.0f;
    Material.Diffuse.g = Material.Ambient.g = 1.0f;
    Material.Diffuse.b = Material.Ambient.b = 1.0f;
    Material.Diffuse.a = Material.Ambient.a = 1.0f;
    pDirect3DDevice -> SetMaterial(&Material);
    D3DXVECTOR3 VectorDir;
    // установим свет
    ZeroMemory(&Light, sizeof(D3DLIGHT9));
    Light.Type = D3DLIGHT_DIRECTIONAL;
    Light.Diffuse.r = 1.0f;
    Light.Diffuse.g = 1.0f;

```

```

    Light.Diffuse.b = 1.0f;
    Light.Range     = 1000.0f;
    // установим нормаль
    VectorDir = D3DXVECTOR3(0.0f, 0.0f, 1.0f),
    D3DXVec3Normalize((D3DXVECTOR3*)&Light.Direction, &VectorDir);
    pDirect3DDevice -> SetLight(0, &Light);
    pDirect3DDevice -> LightEnable(0, TRUE);
    pDirect3DDevice -> SetRenderState(D3DRS_LIGHTING, TRUE);
    pDirect3DDevice -> SetRenderState(D3DRS_AMBIENT, 0);
}
//-----
// функция
// DrawMyText()
// вывод текста на экран
//-----
VOID DrawMyText(LPDIRECT3DDEVICE9 pDirect3DDevice, HFONT hFont,
    char* StrokaTexta, int x, int y, int x1, int y1, D3DCOLOR MyColor)
{
    // создаем шрифт
    hFont = CreateFont(30, 10, 0, 0, FW_NORMAL, FALSE, FALSE, 0, 1, 0,
        0, 0, DEFAULT_PITCH|FF_MODERN, "Arial");
    // координаты прямоугольника
    Rec.left   = x;
    Rec.top    = y;
    Rec.right  = x1;
    Rec.bottom = y1;
    // инициализируем шрифт
    D3DXCreateFont(pDirect3DDevice, hFont, &pFont);
    // начало
    pFont -> Begin();
    // вывод текста
    pFont -> DrawText(StrokaTexta, -1, &Rec, 0, MyColor);
    // конец
    pFont -> End();
}
//-----
// функция
// RenderingDirect3D()
// рисование

```

```

//-----
VOID RenderingDirect3D()
{
    if (pDirect3DDevice == NULL)    // проверяем ошибки
        return;

    pDirect3DDevice -> Clear(0, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
                             D3DCOLOR_XRGB(60, 100, 150), 1.0f, 0);

    // начало сцены
    pDirect3DDevice -> BeginScene();
    // здесь происходит прорисовка сцены
    LightMaterial();
    Matrix();
    pDirect3DDevice -> SetStreamSource(0, pBufferVershin, 0,
                                       sizeof(CUSTOMVERTEX));

    pDirect3DDevice -> SetFVF(D3DFVF_CUSTOMVERTEX);
    pDirect3DDevice -> SetIndices(pBufferIndex);
    pDirect3DDevice -> SetTexture(0, pTextura001);
    pDirect3DDevice -> SetTextureStageState(0, D3DTSS_COLORARG1,
                                           D3DTA_TEXTURE);
    pDirect3DDevice -> SetTextureStageState(0, D3DTSS_COLOROP,
                                           D3DTOP_MODULATE);

    // вывод объекта
    pDirect3DDevice -> DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0,
                                           36, 0, 12);

    // вывод текста
    DrawMyText(pDirect3DDevice, hFont, "Для выхода нажмите клавишу Esc",
               10, 10, 500, 700, D3DCOLOR_ARGB(250, 250, 250, 50));
    pDirect3DDevice -> EndScene();
    pDirect3DDevice -> Present(NULL, NULL, NULL, NULL);
}

//-----
// функция
// DeleteDirect3D()
// освобождение захваченных ресурсов
//-----

VOID DeleteDirect3D()
{

```

```

    if (pTextura001 != NULL)
        pTextura001 -> Release();

    if (pBufferIndex != NULL)
        pBufferIndex -> Release();

    if (pBufferVershin != NULL)
        pBufferVershin -> Release();

    if (pDirect3DDevice != NULL)
        pDirect3DDevice -> Release();

    if (pDirect3D != NULL)
        pDirect3D -> Release();
}
//-----
//  функция
//  MainWinProc()
//  здесь происходит обработка сообщений
//-----

LRESULT CALLBACK MainWinProc(HWND    hwnd,
                                UINT   msg,
                                WPARAM wParam,
                                LPARAM lParam)
{
    switch(msg)
    {
        case WM_DESTROY:
        {
            DeleteDirect3D();
            PostQuitMessage(0);
            return(0);
        }
        case WM_KEYDOWN:
        {
            if (wparam == VK_ESCAPE)
                PostQuitMessage(0);
        }
    }
}

```

[illegible]

```
        GetSystemMetrics(SM_CYSCREEN),
        NULL, NULL, hinstance, NULL)))

return 0;
if (SUCCEEDED(InitialDirect3D(hwnd)))
{
    if (SUCCEEDED(InitialObject() ))
    {
        ShowWindow(hwnd, SW_SHOWDEFAULT);
        UpdateWindow(hwnd);
        ZeroMemory(&msg, sizeof(msg));
        while (msg.message != WM_QUIT)
        {
            if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
            {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
            }
            else
                RenderingDirect3D();
        }
    }
}
return 0;
}
```

Итоги урока

Мы рассмотрели способ текстурирования объекта и теперь имеем возможность на любую поверхность наложить необходимую нам текстуру. Текстурирование поверхностей — это очень важный элемент в работе с графикой, придающий необыкновенный реализм рисуемой сцене. На *уроке 11* мы продолжим знакомство с текстурами, в частности, будет рассмотрено мульти-текстурирование и другие эффекты.

Урок 11



Мультитекстурирование

Этот теоретический урок посвящен работе с текстурами. Весь код этого урока основан на материале *урока 10*. Итоговый листинг рассматриваемого примера вы можете найти на прилагаемом компакт-диске в папке Urok11.

Очевидно, что осуществив текстурирование объекта, т. е. наложив текстуру на примитив один раз, вы можете поверх все той же текстуры наложить еще одну текстуру и так до восьми раз, по отношению к одному объекту. При осуществлении операции по наложению текстур одну на другую у вас появляется возможность добиться различных эффектов поверхности объекта. Само наложение текстуры на текстуру называется *мультитекстурированием*. Мультитекстурирование очень широко применяется в играх. Например, у вас в сцене имеется статичный объект, не имеющий особой ценности, но требующий освещения. Любой расчет света сопряжен с затратами системных ресурсов, а если таких не особо важных объектов несколько штук? Вот тут приходит на помощь мультитекстурирование. С помощью второй текстуры, выполненной в виде тени по аналогии с падающим светом, и в зависимости от предъявляемых требований по освещению этого объекта, вы получаете в итоге полноценно освещенную модель. Фактически на рисунок происходит наложение теней в зависимости от источника света, с той лишь разницей, что накладываемую текстуру вы можете менять сколько угодно, подстраивая ее под себя. Ну, а раз мы коснулись темы освещения, то реализуем данный пример на практике, выполнив мультитекстурирование для освещения объекта.

Поверх текстуры куба наложим еще одну текстуру в виде решетки, получив эффект света, падающего сквозь решетчатый проем.

Перейдем к реализации примера, находящегося на компакт-диске в папке Code\Urok11. В глобальных переменных объявим еще одну переменную, в которой будет храниться вся информация о второй текстуре:

```
LPDIRECT3DTEXTURE9 pTextura002 = NULL;
```

Созданную текстуру в виде файла JPEG сохраним в каталоге данного приложения Urok11 и явно загрузим в проект с помощью функции

`D3DXCreateTextureFromFile()`. Ее прототип подробно разбирался на *уроке 10*:

```
if (FAILED(CreateTextureFromFile(pDirect3DDevice, "Reshetka.jpg",
                                &pTextura002))) ;

return E_FAIL;
```

Далее необходимо с помощью функции `IDirect3DDevice9::SetTexture` установить соответствующий идентификатор или назначить текстурный уровень для второй текстуры. В функции `RenderingDirect3D()` напомним:

```
pDirect3DDevice -> SetTexture(1, &pTextura002);
```

Первый параметр и есть текстурный уровень для второй текстуры, адрес которой содержится во втором параметре. И завершающий шаг — это определение текстурного состояния путем вызова функции `SetTextureStageState`, параметры которой также рассматривались на *уроке 10*.

```
pDirect3DDevice -> SetTextureStageState(1, D3DTSS_TEXCOORDINDEX, 0);
```

По сравнению с предыдущим уроком добавился еще один вызов функции `IDirect3DDevice9::SetTextureStageState` с использованием параметра `D3DTSS_TEXCOORDINDEX`. Данный параметр указывает, что наложение текстуры следует производить на определенные координаты. Значение 0 в третьем параметре как раз и подразумевает первый текстурный уровень, т. е. вторая текстура накладывается на первую текстуру на те же самые координаты.

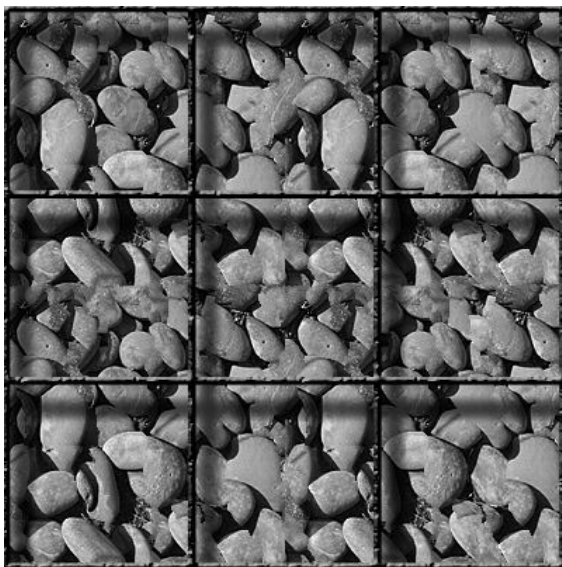


Рис. 11.1. Мультитекстурирование

Последующие вызовы функции `IDirect3DDevice9::SetTextureStageState` аналогичны вызовам, осуществлявшимся на прошлом уроке:

```
pDirect3DDevice -> SetTextureStageState(1, D3DTSS_COLORARG1,  
                                         D3DTA_TEXTURE);  
pDirect3DDevice -> SetTextureStageState(1, D3DTSS_COLOROP,  
                                         D3DTOP_MODULATE);
```

На рис. 11.1 представлен вариант наложения текстур друг на друга.

Цветовые ключи

Рассмотрим еще одну операцию, имеющую название *цветовой ключ* (color key). Цветовые ключи широко применяются в играх. Скажем, у вас есть некая текстура, нарисованная на каком-то однотонном фоне, черном или белом, не важно. Ваша цель наложить эту текстуру на объект, но так, чтобы основной рисунок отображался, а весь остальной однотонный фон — нет. То есть необходимо отразить основную текстуру на объекте, а окружающий ее фон скрыть. Для этих целей существуют цветовые ключи, с помощью которых вы задаете числовое значение, например, для цвета фона, и для такого ключа весь фон не рисуется.

Прежде чем перейти к рассмотрению примера цветовых ключей, стоит познакомиться с элементом, имеющим название *альфа-блендинг* (alpha blending). Альфа-блендинг необходим при проецировании прозрачных и полупрозрачных пикселей. Цвет в Direct3D задается в цветовых системах RGB и ARGB:

- A — альфа-цвет;
- R — красный цвет;
- G — зеленый цвет;
- B — синий цвет.

Альфа-цвет или альфа-канал (alpha channel) нужен для создания прозрачности (transparency), с ней-то мы и столкнемся в цветовых ключах. Чтобы включить альфа-блендинг, нужно воспользоваться функцией `SetRenderState()` с соответствующим ключом:

```
pDirect3DDevice -> SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE)
```

Перейдем к цветовым ключам.

Работа с цветовыми ключами абсолютно идентична текстурированию, с той лишь разницей, что когда необходимо загрузить текстуру из файла, используется специально предназначенная для этого функция

`D3DXCreateTextureFromFileEx()`, в которой несколько больше параметров, чем у ее предшественницы `D3DXCreateTextureFromFile()`. А также имеется параметр `ColorKey`, отвечающий за цветовой ключ. Рассмотрим более подробно прототип функции `D3DXCreateTextureFromFileEx()`:

```
HRESULT D3DXCreateTextureFromFileEx(
```

```
    LPDIRECT3DDEVICE9    pDevice,
    LPCTSTR              pSrcFile,
    UINT                  Width,
    UINT                  Height,
    UINT                  MipLevels,
    DWORD                 Usage,
    D3DFORMAT             Format,
    D3DPOOL               Pool,
    DWORD                 Filter,
    DWORD                 MipFilter,
    D3DCOLOR              ColorKey,
    D3DXIMAGE_INFO*       pSrcInfo,
    PALETTEENTRY*         pPalette,
    LPDIRECT3DTEXTURE9*   ppTexture
```

```
);
```

Функция `D3DXCreateTextureFromFileEx()` имеет следующие параметры:

- ❑ `pDevice` — указатель на интерфейс `IDirect3DDevice9`, наше устройство `Direct3D`, связанное с текстурой;
- ❑ `pSrcFile` — указатель на строку, обозначает имя файла загружаемой текстуры;
- ❑ `Width` — ширина текстуры в пикселах. Есть возможность устанавливать нулевое значение или флаг `D3DX_DEFAULT` для обработки значения по умолчанию;
- ❑ `Height` — высота текстуры в пикселах. Можно установить значение 0 или применить флаг `D3DX_DEFAULT`;
- ❑ `MipLevels` — "мип-уровень" используется в "мип-маппинге" (mip-mapping) — это один из видов фильтрации текстур. Пока этот параметр использовать не будем, установим нулевое значение или флаг `D3DX_DEFAULT`;
- ❑ `Usage` — это формат используемой поверхности. Сейчас он не задействован, поэтому поставим значение 0;

- ❑ **Format** — член перечисляемого типа `D3DFORMAT`. Текстура может иметь различный формат, поэтому формат необходимо задавать вручную либо воспользоваться флагом `D3DEMT_UNKNOWN` — значение по умолчанию. Этот флаг сам определит формат текстуры и передаст его в функцию;
- ❑ **Pool** — описывает класс памяти, в который данная текстура будет загружена. Член перечисляемого типа `D3DPOOL`;
- ❑ **Filter** — фильтр, определяющий фильтрацию текстуры. Здесь можно выставить флаги управления из `D3DX_FILTER`. Значение по умолчанию `D3DX_DEFAULT`, им и воспользуемся;
- ❑ **MipFilter** — фильтр, определяющий фильтрацию текстур в "мип-маппинге". Используются флаги управления из `D3DX_FILTER`, значение по умолчанию — `D3DX_DEFAULT`;
- ❑ **ColorKey** — это и есть цветовой ключ, в котором определяется нерисуемый цвет. Значение `D3DCOLOR` всегда 32-битное ARGB, альфа-канал устанавливается в FF. Например, если вы хотите не прорисовывать черный цвет, то значение должно быть равно `0xFF000000`. Когда в этом параметре применяется значение, равное 0, то цветовой ключ будет отключен;
- ❑ **pSrcInfo** — указатель на структуру `D3DXIMAGE_INFO`. Эта структура заполняется данными об исходном файле текстуры. Значение `NULL` устанавливается по умолчанию;
- ❑ **pPalette** — указатель на структуру `PALETTEENTRY`, который определяет 256-цветную палитру. Для заполнения можно установить значение `NULL`. Обе структуры — `D3DXIMAGE_INFO` и `PALETTEENTRY` — рассмотрим сразу после окончания разбора параметров;
- ❑ **ppTexture** — адрес указателя на интерфейс `IDirect3DTexture9`, результат всей операции.

Функция `D3DXCreateTextureFromFileEx()` поддерживает следующие расширения загружаемых текстур: `bmp`, `dds`, `dib`, `jpg`, `png`, `tga`.

Общий вид функции `D3DXCreateTextureFromFileEx()` может быть таким:

```
D3DXCreateTextureFromFileEx(pDirect3DDevice, "Текстура.bmp"  
                             0, 0, 0, 0, D3DEMT_UNKNOWN, D3DPOOL_DEFAULT,  
                             D3DX_DEFAULT, D3DX_DEFAULT, 0xFF000000,  
                             NULL, NULL, &Texture002);
```

При помощи вызова функции `SetRenderState()` можно включить альфа-блендинг, который допускает альфа-смешивание:

```
pDirect3DDevice -> SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
```

А вот как определяются состояние смешивания и адресат смешивания:

```
pDirect3DDevice -> SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);  
pDirect3DDevice -> SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
```

Остальной код аналогичен примеру по текстурированию, рассмотренному на уроке 10.

Итоги урока

На этом уроке мы рассмотрели мультитекстурирование и цветовые ключи. Все приемы применяются при программировании игр. *Урок 12* мы посвятим загрузке моделей, сделанных в 3D-редакторах, в наше приложение.

Урок 12



Загрузка X-файлов

До этого момента мы создавали простые фигуры, состоящие из небольшого количества вершин, например, в квадрате их было 6, а в кубе 36. Но как быть, когда необходимо создать действительно сложный примитив? Не забывайте: любой объект строится из набора треугольников, которые в свою очередь состоят из заданного количества вершин. Вспомните *урок 2*, где после инсталляции DirectX 9 SDK был создан демонстрационный пример, а в качестве фигуры был представлен чайник, содержащий в себе 1178 вершин, при том, что это не самый сложный объект! Тогда бы перспектива стать программистом компьютерных игр выглядела бы не такой уж и радостной. Но на самом деле не все так сложно, как может показаться. Необходимо просто воспользоваться практически любым 3D-пакетом, создающим трехмерные модели, а потом загрузить эту модель с помощью средств, имеющихся в DirectX, в свое приложение. Само создание трехмерного объекта в этой книге не рассматривается, да и вообще-то это и не работа программиста. Но могу посоветовать неплохую книгу издательства "БХВ-Петербург" "3DS Max 5: от фантазии к реальности" Б. Кулагина.

3DS Max является лидером в этом направлении, и большинство моделей для игр делаются с помощью этого пакета. Но для нас, программистов, главная задача — загрузка готовой трехмерной модели в игру. Сама по себе эта задача многогранна и изобилует большим количеством способов реализации. Мы остановимся на стандартном способе, предлагаемом в DirectX 9 SDK, а именно, загрузке X-файла в ваше приложение посредством все того же стандартного кода.

X-файл — это файл, содержащий в себе трехмерный объект, параметры материалов, нормали, название текстур, анимации и многие другие значения, которые вы можете загружать в приложение. X-файл по своей сути обыкновенный X-формат, имеющий свое расширение, скажем, *cube.x*, который создается путем конвертации 3D-модели. Утилит для конвертации большое множество, как бесплатных, так и платных. Создав в том же 3DS Max модель, вы конвертируете ее в X-формат и загружаете в свое приложение. В составе

DirectX 9 имеется программа Mesh Viewer, с помощью которой можно слегка модифицировать X-файл. О загрузке X-файла мы и поговорим.

В качестве модели к этому уроку был использован X-файл из DirectX 9 SDK с изображением тигра. Он используется во всех примерах в документации к DirectX 9 SDK как подопытный, а поскольку я не "3D-модельер", то я воспользовался этой моделью.

В DirectX 9 есть интерфейс, имеющий интересное название *мэш* (Mesh) — ID3DMesh, что в переводе означает сетка. Если представить любую модель в виде каркаса треугольников, то выглядеть она будет как сетка, возможно, поэтому и было дано такое название. С помощью этого интерфейса, а точнее определенных в его составе структур, функций и макросов происходит работа с X-файлами.

Создайте новый проект и скопируйте код с предыдущего *урока 11*. Сам код претерпит большие изменения, и сначала уберем из него все то, что уже не понадобится при работе с X-файлами. А не понадобятся нам указатели, отвечающие за вершинный и индексный буферы, указатели текстур, материала и света, а также все функции, связанные с созданием объекта. Мы загрузим X-файл, а он уже содержит данные о текстурах и материале. Перейдем непосредственно к самому коду и удалим ненужные элементы.

В функции `InitialDirect3D()` оставим почти все без изменения и только удалим такую строку:

```
pDirect3DDevice -> SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);
```

Такая запись отвечала за отсечение невидимых граней объекта (при работе с X-файлами все происходит несколько иначе). Следующую функцию `InitialObject()`, с ее помощью мы создавали объект, удаляем полностью, на ее место придет новая функция, загружающая X-файл. Функция `Matrix()` остается без изменения. Можете изменить мировую матрицу, задав, например другой угол сдвига, или поменять точку просмотра камеры посредством изменения значений в матрице вида. Функция `DrawMyText()` тоже остается без изменения. А вот функция `LightMaterial()`, инициализирующая свет и материал, пока не понадобится, поскольку данные о материале содержатся в X-файле и загружаются непосредственно из этого файла. В функции `RenderingDirect3D()` между началом сцены `BeginScene()` и концом сцены `EndScene()` уберем все, оставив только два вызова функции `Matrix()` и `DrawMyText()`. Прорисовкой загружаемого X-файла будет заниматься своя функция. И в конце удалите из функции `DeleteDirect3D()` неиспользуемые указатели элементов, которые мы только что удалили.

Тем самым мы избавились от всего, что было связано с инициализацией и прорисовкой объекта, теперь приступим к загрузке и отображению X-файла на экране монитора.

Для работы с X-файлом понадобится заголовочный файл `d3dx9mesh.h`, подключите его в создаваемое приложение. Перейдем в глобальные переменные, где создадим ряд указателей на необходимые нам интерфейсы:

```
LPD3DXMESH pMesh          = NULL;
LPD3DXBUFFER pMeshBuffer  = NULL;
D3DMATERIAL9* pMeshMaterial = NULL;
LPDIRECT3DTEXTURE9* pMeshTexture = NULL;
DWORD dwNumber;
```

В первой строке создается указатель на интерфейс `ID3DMesh`, отвечающий за работу с мэшем. Второй строкой создается указатель `pMeshBuffer` на интерфейс `ID3DXBuffer`. Это материальный буфер, содержащий информацию о материале и текстуре для X-файла. В двух следующих строках, как видно из названий, создаются два указателя для работы с материалом и текстурой. В этих двух указателях будут храниться материал и текстура для объекта. И последняя переменная — `dwNumber` — предназначена для хранения материала в загружаемом X-файле.

Следующий шаг состоит в создании функции, с помощью которой будет происходить загрузка и инициализация объекта. Назовем ее `InitialMesh()`. Реализация этой функции выглядит следующим образом:

```
HRESULT InitialMesh()
{
    if (FAILED(D3DXLoadMeshFromX("Mesh.x", D3DXMESH_SYSTEMMEM,
        pDirect3DDevice, NULL, &pMeshBuffer, NULL,
        &dwNumber, &pMesh)))
        return E_FAIL;
    D3DXMATERIAL* D3DXMeshMaterials;
    D3DXMeshMaterials = (D3DXMATERIAL *)pMeshBuffer -> GetBufferPointer();

    pMeshMaterials = new D3DMATERIAL9[dwNumber];
    pMeshTexture    = new LPDIRECT3DTEXTURE9[dwNumber];
    for (DWORD i = 0; i < dwNumber; i++)
    {
        // копируем материал
        pMeshMaterials[i] = D3DXMeshMaterials[i].MatD3D;
        // устанавливаем окружающий свет
        pMeshMaterials[i].Ambient = pMeshMaterials[i].Diffuse;
        // загружаем текстуру
        if (FAILED (D3DXCreateTextureFromFile(pDirect3DDevice,
            D3DXMeshMaterials[i].pTextureFilename, &pMeshTexture[i])))
```

```

    pMeshTextura[i]=NULL;
}
// уничтожаем материальный буфер
pMeshBuffer -> Release();
return S_OK;
}

```

В начале работы функции `InitialMesh()` происходит вызов функции `D3DXLoadMeshFromX()`, с ее помощью осуществляется загрузка X-файла. Рассмотрим ее прототип:

```

HRESULT D3DXLoadMeshFromX(
    LPCTSTR          pFilename,
    DWORD            Options,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXBUFFER*     ppAdjacency,
    LPD3DXBUFFER*     ppMaterials,
    LPD3DXBUFFER*     ppEffectInstances,
    DWORD*            pNumMaterials,
    LPD3DXMESH*       ppMesh
);

```

Функция `D3DXLoadMeshFromX()` имеет следующие параметры:

- ❑ `pFilename` — указатель на строку, в которой приведено имя загружаемого X-файла;
- ❑ `Options` — член перечисляемого типа `D3DXMESH`, определяет комбинацию одного или более флагов для различных вариантов создания мэша. Рассмотрим несколько из имеющихся флагов:
 - `D3DXMESH_VB_SYSTEMMEM` — этот флаг необходимо использовать с флагом `D3DPOOL_SYSTEMMEM` для определения класса памяти буфера вершин;
 - `D3DXMESH_VB_MANAGED` — применяется с флагом `D3DPOOL_MANAGED` для определения класса памяти буфера вершин;
 - `D3DXMESH_IB_SYSTEMMEM` — этот флаг необходимо использовать с флагом `D3DPOOL_SYSTEMMEM` для определения памяти индексного буфера;
 - `D3DXMESH_IB_MANAGED` — применяется с флагом `D3DPOOL_MANAGED` для определения памяти индексного буфера;
 - `D3DXMESH_USEHWONLY` — используется для обработки только аппаратных средств;
 - `D3DXMESH_MANAGED` — эквивалент комбинации двух флагов `D3DXMESH_VB_MANAGED` и `D3DXMESH_IB_MANAGED`;

- `D3DXMESH_SYSTEMMEM` — этим флагом мы воспользовались в нашем примере. Он эквивалентен комбинации двух флагов `D3DXMESH_VB_SYSTEMMEM` и `D3DXMESH_IB_SYSTEMMEM`;
- `pDevice` — это указатель `pDirect3DDevice` на интерфейс `IDirect3DDevice9`, связывающий X-файл с устройством `Direct3D`;
- `ppAdjacency` — адрес указателя на буфер, содержащий данные о смежности. В этом параметре используется указатель на интерфейс `ID3DXBuffer`. В нашем примере этот указатель не требуется, поэтому поставим `NULL`;
- `ppMaterials` — адрес указателя на интерфейс `ID3DXBuffer`. По возвращению функции этот параметр, а у нас `pMeshBuffer`, заполнится данными структуры `D3DXMATERIAL` для X-файла;
- `ppEffectInstances` — адрес указателя на интерфейс `ID3DXBuffer`, содержащий специфические эффекты. Этот параметр не используется, значение установлено в `NULL`;
- `pNumMaterials` — указатель на массив данных для структуры `D3DXMATERIAL`. Мы используем переменную `dwNumber`;
- `ppMesh` — адрес указателя на интерфейс `ID3DXMesh`, результат всей операции.

Рассмотренная функция `D3DXLoadMeshFromX()`, загружает в приложение X-файл. Информация, возвращаемая данной функцией, содержит данные о материале и текстуре. Эти свойства необходимо извлечь с помощью материального буфера, посредством вызова функции `ID3DXBuffer::GetBufferPointer`. Функция `GetBufferPointer()` не имеет параметров и возвращает указатель на буфер данных:

```
D3DXMATERIAL* D3DXMeshMaterials;
```

```
D3DXMeshMaterials = (D3DXMATERIAL *)pMeshBuffer -> GetBufferPointer();
```

В этой строке кода создается указатель на структуру `D3DXMATERIAL`, которой присваивается адрес указателя на материальный буфер. После чего необходимо создать массив для хранения свойств материала и текстуры, который заполнится данными путем извлечения этих свойств. Весь массив данных для материала и текстуры создается в области динамического обмена. Напоминаю, что память разделена на несколько областей:

- регистровая память;
- стековая память;
- сегментная память;
- область глобальных переменных;
- динамическая память.

Регистровая память содержит внутренние данные программы, в стековой памяти обычно размещаются локальные переменные и параметры функций.

Сегментная память хранит программный код, а область глобальных переменных содержит глобальные переменные. Остальная область памяти является динамической и распределяется между разными объектами. В область динамического распределения можно также включить область глобальных переменных. Чтобы выделить память в области динамического обмена, используется ключевое слово `new`, после чего необходимо указать тип объекта, который и размещается в области динамического обмена. Такая операция определяет размер памяти для хранения объекта. В итоге оператор `new` возвратит адрес выделенной области памяти.

Выделим область памяти для хранения материала и текстуры:

```
pMeshMaterials = new D3DMATERIAL9[dwNumber];
pMeshTextura   = new LPDIRECT3DTEXTURE9[dwNumber];
```

В выделенную область памяти мы поместим параметры материала и текстуры для объекта из X-файла. Поскольку используется массив данных, то логично применить цикл `for`, шаг за шагом заполняющий весь массив необходимыми данными:

```
for (DWORD i = 0; i < dwNumber; i++)
{
    pMeshMaterials[i] = D3DXMeshMaterials[i].MatD3D;
    pMeshMaterials[i].Ambient = pMeshMaterials[i].Diffuse;
    if (FAILED(D3DXCreateTextureFromFile(pDirect3DDevice,
        D3DXMeshMaterials[i].pTextureFilename, &pMeshTextura[i])))
        pMeshTextura[i] = NULL;
}
```

В первой строке кода цикла `for` происходит копирование материала в выделенную область памяти при использовании переменной `MatD3D`. Эта переменная является членом структуры `D3DXMATERIAL`. Обращение к этому члену посредством оператора точки возможно, поскольку переменная `D3DXMeshMaterials` является объектом структуры `D3DXMATERIAL`. Сама структура несложная и выглядит следующим образом:

```
tupedef struct D3DXMATERIAL{
    D3DMATERIAL9 MatD3D;
    LPSTR        pTextureFilename;
} D3DXMATERIAL;
```

Структура `D3DXMATERIAL` имеет такие параметры:

- `MatD3D` — член структуры `D3DMATERIAL9`, описывающий материальные свойства;
- `pTextureFilename` — указатель на строку, определяющий имя файла текстуры.

Сам тип `LPD3DXMATERIAL` определен как указатель на структуру `D3DXMATERIAL`.

Вторая строка кода цикла `for` устанавливает окружающее освещение материала. Оба шага по копированию материала и установке окружающего освещения абсолютно идентичны действиям, применяемым при работе со светом и материалом на предыдущих уроках. Сейчас изменился только формат записи, поскольку используется большой массив данных.

После чего следует вызов функции `D3DXCreateTextureFromFile()`, с которой мы уже хорошо знакомы. Она загружает текстуру для объекта из X-файла.

И в конце всей функции `InitialMesh()` удалим материальный буфер:

```
pMeshBuffer -> Release();
```

Объект, наряду с текстурой и материалом, загружен, и его можно выводить на экран монитора. За прорисовку будет отвечать функция `DrawMyMesh()`, которую мы сейчас и напишем:

```
VOID DrawMyMesh()
{
    for (DWORD i = 0; i < dwNumber; i++)
    {
        pDirect3DDevice -> SetMaterial(&pMeshMaterials[i]);
        pDirect3DDevice -> SetTexture(0, pMeshTextura[i]);
        pMesh -> DrawSubset(i);
    }
}
```

Функция `DrawMyMesh()` сама по себе не сложная и состоит из цикла `for`, внутри которого происходит установка материала и текстуры для объекта посредством вызова функций: `SetMaterial()` — для материала и `SetTexture()` — для текстуры. В качестве параметров используются указатели на массивы данных. Весь цикл заканчивается вызовом функции `ID3DXBaseMesh::DrawSubset`, с помощью которой происходит прорисовка загруженного X-файла. Прототип функции выглядит так:

```
HRESULT DrawSubset(DWORD AttribId);
```

Единственный параметр `AttribId` — это атрибут, рисующий установленную часть мэша. Весь мэш рисуется небольшими областями данных посредством задания индекса атрибута.

Вызов функции `DrawMyMesh()` необходимо произвести в функции `RenderingDirect3D()`, между началом сцены `BeginScene()` и концом сцены `EndScene()`, для вывода X-файла на экран монитора. И еще нужно позабо-

таться об установке окружающего освещения путем вызова функции `SetRenderState()` в функции `InitialDirect3D()`:

```
pDirect3DDevice -> SetRenderState(D3DRS_AMBIENT, 0xffffffff);
```

На этом все действия по загрузке, инициализации и выводу X-файла на экран завершаются. Осталось только в функции `DeleteDirect3D()` освободить все захваченные ресурсы. Код этой функции приведен в листинге 12.1. Замечу только, что при использовании динамической памяти, выделенной с помощью оператора `new`, ее удаление должно происходить оператором `delete`.

Текст файла `Mesh.cpp` приведен в листинге 12.1.

Листинг 12.1. Файл `Mesh.cpp`

```
//-----
// Mesh.cpp
// загружаем X-файл
//-----
#include <windows.h>    // подключаем заголовочный файл Windows
#include <d3d9.h>        // подключаем заголовочный файл DirectX 9 SDK
#include <d3dx9.h>       // подключаем библиотеку из D3DX-утилит
                        // для работы с матрицами
#include <mmsystem.h>    // подключаем системную библиотеку
#include <d3dx9core.h>   // подключаем системный заголовочный файл
#include <d3dx9mesh.h>   // подключаем заголовочный файл
                        // для работы с мэшем
//-----
// глобальные переменные
//-----
LPDIRECT3D9 pDirect3D = NULL;           // главный 3D-объект
LPDIRECT3DDEVICE9 pDirect3DDevice = NULL; // устройство
LPD3DXMESH pMesh = NULL;               // мэш
LPD3DXBUFFER pMeshBuffer = NULL;       // материальный буфер
D3DMATERIAL9* pMeshMaterials = NULL;   // материал
LPDIRECT3DTEXTURE9* pMeshTextura = NULL; // текстура для мэша
DWORD dwNumber;                        // индекс
LPD3DXFONT pFont = NULL;               // шрифт Direct3D
RECT Rec;                              // прямоугольник
HFONT hFont;                           // шрифт
```

```
//-----  
//  функция  
//  IDirect3D()  
//  инициализация Direct3D  
//-----  
HRESULT IDirect3D(HWND hwnd)  
{  
    if ( NULL == (pDirect3D = Direct3DCreate9(D3D_SDK_VERSION)) )  
        return E_FAIL;  
  
    D3DDISPLAYMODE Display;  
    if (FAILED(pDirect3D -> GetAdapterDisplayMode(  
                                                D3DADAPTER_DEFAULT, &Display)) )  
        return E_FAIL;  
  
    D3DPRESENT_PARAMETERS Direct3DParametr;  
    ZeroMemory(&Direct3DParametr, sizeof(Direct3DParametr));  
    Direct3DParametr.Windowed = FALSE;  
    Direct3DParametr.SwapEffect = D3DSWAPEFFECT_DISCARD;  
    Direct3DParametr.BackBufferFormat = Display.Format;  
    Direct3DParametr.EnableAutoDepthStencil = TRUE;  
    Direct3DParametr.AutoDepthStencilFormat = D3DFMT_D16;  
    Direct3DParametr.BackBufferWidth = GetSystemMetrics(SM_CXSCREEN);  
    Direct3DParametr.BackBufferHeight = GetSystemMetrics(SM_CYSCREEN);  
    Direct3DParametr.BackBufferCount = 3;  
    Direct3DParametr.FullScreen_RefreshRateInHz = Display.RefreshRate;  
  
    if (FAILED(pDirect3D -> CreateDevice(D3DADAPTER_DEFAULT,  
                                         D3DDEVTYPE_HAL, hwnd,  
                                         D3DCREATE_SOFTWARE_VERTEXPROCESSING,  
                                         &Direct3DParametr, &pDirect3DDevice)) )  
        return E_FAIL;  
    //  окружающее освещение  
    pDirect3DDevice -> SetRenderState(D3DRS_AMBIENT, 0xffffffff);  
    //  подключаем Z-буфер  
    pDirect3DDevice -> SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);  
    return S_OK;  
}
```

```

//-----
//  функция
//  InitialMesh()
//  инициализация меша
//-----
HRESULT InitialMesh()
{
    if (FAILED(D3DXLoadMeshFromX("Mesh.x", D3DXMESH_SYSTEMMEM,
                                pDirect3DDevice, NULL, &pMeshBuffer,
                                NULL, &dwNumber, &pMesh)))

        return E_FAIL;

    D3DXMATERIAL* D3DXMeshMaterials;
    D3DXMeshMaterials = (D3DXMATERIAL *)pMeshBuffer -> GetBufferPointer();
    pMeshMaterials = new D3DMATERIAL9[dwNumber];
    pMeshTextura = new LPDIRECT3DTEXTURE9[dwNumber];
    for (DWORD i = 0; i < dwNumber; i++)
    {
        // копируем материал
        pMeshMaterials[i] = D3DXMeshMaterials[i].MatD3D;
        // устанавливаем окружающий свет
        pMeshMaterials[i].Ambient = pMeshMaterials[i].Diffuse;
        // загружаем текстуру
        if (FAILED(D3DXCreateTextureFromFile(pDirect3DDevice,
                                            D3DXMeshMaterials[i].pTextureFilename, &pMeshTextura[i])))
            pMeshTextura[i] = NULL;
    }
    // уничтожаем материальный буфер
    pMeshBuffer -> Release();
    return S_OK;
}
//-----
//  функция
//  DrawMyMesh()
//  рисует созданный объект
//-----
VOID DrawMyMesh()
{
    for (DWORD i = 0; i < dwNumber; i++)
    {

```

```

    // устанавливаем материал и текстуру
    pDirect3DDevice -> SetMaterial(&pMeshMaterials[i]);
    pDirect3DDevice -> SetTexture(0, pMeshTextura[i]);
    // рисуем мэш
    pMesh -> DrawSubset(i);
}
}
//-----
// функция
// Matrix()
// мировая матрица, матрица вида, матрица проекции
//-----
VOID Matrix()
{
    D3DXMATRIX MatrixWorld,      // мировая матрица
    D3DXMATRIX MatrixView;       // матрица вида
    D3DXMATRIX MatrixProjection; // матрица проекции
    // MatrixWorld
    UINT Time = timeGetTime() % 5000;
    FLOAT Angle = Time * (2.0f * D3DX_PI) / 5000.0f;
    D3DXMatrixRotationY(&MatrixWorld, Angle);
    pDirect3DDevice -> SetTransform(D3DTS_WORLD, &MatrixWorld);
    // MatrixView
    D3DXMatrixLookAtLH(&MatrixView, &D3DXVECTOR3(0.0f, 0.0f, -8.0f),
                      &D3DXVECTOR3(0.0f, 0.0f, 0.0f),
                      &D3DXVECTOR3(0.0f, 1.0f, 0.0f));
    pDirect3DDevice -> SetTransform(D3DTS_VIEW, &MatrixView);
    // MatrixProjection
    D3DXMatrixPerspectiveFovLH(&MatrixProjection, D3DX_PI/4,
                              1.0f, 1.0f, 100.0f);
    pDirect3DDevice -> SetTransform(D3DTS_PROJECTION, &MatrixProjection);
}
//-----
// функция
// DrawMyText()
// выводит текст на экран
//-----
VOID DrawMyText(LPDIRECT3DDEVICE9 pDirect3DDevice, HFONT hFont,
               char* StrokaTexta, int x, int y, int x1,

```

```

        int y1, D3DCOLOR MyColor)
{
    // создаем шрифт
    hFont = CreateFont(30, 10, 0, 0, FW_NORMAL, FALSE, FALSE, 0, 1, 0,
                      0, 0, DEFAULT_PITCH|FF_MODERN, "Arial");

    // координаты прямоугольника
    Rec.left   = x;
    Rec.top    = y;
    Rec.right  = x1;
    Rec.bottom = y1;

    // инициализируем шрифт
    D3DXCreateFont(pDirect3DDevice, hFont, &pFont);
    // начало
    pFont->Begin();
    // вывод текста
    pFont -> DrawText(StrokaTexta, -1, &Rec, 0, MyColor);
    // конец
    pFont -> End();
}

//-----
// RenderingDirect3D()
// рисование
//-----
VOID RenderingDirect3D()
{
    if (pDirect3DDevice == NULL)    // проверяем ошибки
        return;

    // очистим задний буфер
    pDirect3DDevice -> Clear(0, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
                           D3DCOLOR_XRGB(60, 100, 150), 1.0f, 0);

    // начало сцены, здесь происходит прорисовка сцены
    pDirect3DDevice -> BeginScene();
    // матрицы
    Matrix();
    // рисуем мэш
    DrawMyMesh();
    // вывод текста
    DrawMyText(pDirect3DDevice, hFont, "Урок 12", 10, 10, 500,
              700, D3DCOLOR_ARGB(250, 250, 250, 50));
}

```

```
// конец сцены
pDirect3DDevice -> EndScene();
// выводим на экран
pDirect3DDevice -> Present(NULL, NULL, NULL, NULL);
}
```

```
//-----
```

```
// Функция
// DeleteDirect3D()
// освобождает захваченные ресурсы
//-----
```

```
VOID DeleteDirect3D()
{
    if (pMeshMaterials != NULL)
        delete[] pMeshMaterials;

    if (pMeshTextura)
    {
        for (DWORD i = 0; i < dwNumber; i++)
        {
            if (pMeshTextura[i])
                pMeshTextura[i] -> Release();
        }
        delete[] pMeshTextura;
    }
    if (pMesh != NULL)
        pMesh->Release();

    if (pDirect3DDevice != NULL)
        pDirect3DDevice -> Release();

    if (pDirect3D != NULL)
        pDirect3D -> Release();
}
```

```
//-----
```

```
// MainWinProc()
// здесь происходит обработка сообщений
//-----
```

```
LRESULT CALLBACK MainWinProc(HWND hwnd,
                               UINT msg,
```

```

        WPARAM  wparam,
        LPARAM  lparam)
{
    switch (msg)
    {
        case WM_DESTROY:
        {
            DeleteDirect3D();
            PostQuitMessage(0);
            return(0);
        }
        case WM_KEYDOWN:
        {
            if (wparam == VK_ESCAPE)
                PostQuitMessage(0);
            return 0;
        }
    }
    return DefWindowProc(hwnd, msg, wparam, lparam);
}

//-----
//  функция
//  WinMain
//  входная точка приложения
//-----

int WINAPI WinMain(HINSTANCE  hinstance,
                   HINSTANCE  hprevinstance,
                   HINSTANCE  hprevinstance,
                   LPSTR       lpcmdline,
                   int         ncmdshow)
{
    WNDCLASSEX windowclass;           // создаем класс
    HWND        hwnd;                 // создаем дескриптор окна
    MSG         msg;                   // идентификатор сообщения

    // определим класс окна WNDCLASSEX
    windowclass.cbSize = sizeof(WNDCLASSEX);
    windowclass.style = CS_DBLCLKS|CS_OWNDC|CS_HREDRAW|CS_VREDRAW;
    windowclass.lpfnWndProc = MainWinProc;

```

```
windowsclass.cbClsExtra = 0;
windowsclass.cbWndExtra = 0;
windowsclass.hInstance = hinstance;
windowsclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
windowsclass.hCursor = LoadCursor(NULL, IDC_ARROW);
windowsclass.hbrBackground = (HBRUSH)GetStockObject (GRAY_BRUSH);
windowsclass.lpszMenuName = NULL;
windowsclass.lpszClassName = "WINDOWSCCLASS";
windowsclass.hIconSm = LoadIcon (NULL, IDI_APPLICATION);

// регистрируем класс
if (!RegisterClassEx(&windowsclass))
return (0);

// теперь, когда класс зарегистрирован, можно создать окно

if (!(hwnd = CreateWindowEx(NULL,           // стиль окна
    "WINDOWSCCLASS",           // класс
    0,                          // название окна
    WS_OVERLAPPEDWINDOW|WS_VISIBLE,
    0,0,                        // левый верхний угол
    GetSystemMetrics(SM_CXSCREEN), // ширина
    GetSystemMetrics(SM_CYSCREEN), // высота
    NULL,                       // дескриптор родительского окна
    NULL,                       // дескриптор меню
    hinstance,                 // дескриптор экземпляра приложения
    NULL)))                    // указатель на данные окна
return 0;

if (SUCCEEDED(InitialDirect3D(hwnd)))
{
    if (SUCCEEDED(InitialMesh() ))
    {
        ShowWindow(hwnd, SW_SHOWDEFAULT);
        UpdateWindow(hwnd);

        ZeroMemory(&msg, sizeof(msg));
        while (msg.message != WM_QUIT)
        {
```

```
        if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else
            RenderingDirect3D();
    }

}

return 0;
}
```

Итоги урока

На этом уроке была показана общая модель загрузки X-файлов из приложения. Создать объект средствами Direct3D можно, но проще создать модель в 3D-редакторе и воспользоваться механизмом, имеющимся в DirectX для загрузки объекта. В современных играх подчас целые уровни загружаются в приложение, используя подобную технологию, а также другие мощные механизмы работы.

Урок 13



Вершинные и пиксельные шейдеры

Это последний урок, посвященный Direct3D, в котором вы познакомитесь с очень мощным процессом обработки данных. Но прежде чем перейти к теме урока, хотелось бы объединить все, что было изучено на прошлых уроках, для того чтобы более четко уяснить необходимость использования вершинных шейдеров.

Когда вы только начинаете программирование трехмерной графики, самое первое, что нужно сделать, — это создать оконное приложение, в которое впоследствии вы будете интегрировать свой код. Далее происходит трудоемкий процесс по инициализации, настройке и созданию устройства Direct3D, на платформе которого и происходит проектирование трехмерной сцены. Вся трехмерная сцена состоит из массы разнородных объектов, построение которых происходит при помощи полигонов или треугольников. Создание одного треугольника в пространстве происходит с помощью вершин, координаты которых задаются по осям X , Y и Z . Для того чтобы прорисовать объект на дисплее, все вершины проходят через определенный конвейер рендеринга, где происходит своего рода сборка объекта. Весь конвейер состоит из большого количества механизмов и всевозможных операций. На *уроке 2* была представлена обобщенная модель конвейера, но большинство моментов было умышленно опущено, поскольку вы тогда только делали первые шаги в DirectX и не имели ни малейшего представления о текстурировании, освещении, Z -буфере, матричных преобразованиях и всем том, что изучили на прошлых уроках. В тот момент, наверное, вы многое не поняли бы, а вот сейчас самое время.

Графический конвейер

Как вы знаете, архитектура Direct3D построена специальным образом, что дает возможность работать с аппаратным обеспечением компьютера напрямую, минуя стандартные сервисы Win32. Такая возможность достигается с

помощью аппаратного уровня абстракции и драйвера устройства. Если мы говорим о графике, то, очевидно, что таким устройством является видеоадаптер, установленный у вас в компьютере и имеющий свой графический процессор, отвечающий за обработку графики. Но в целом видеоадаптер не понимает, что вы хотите вывести на экран, допустим, фотографию полуобнаженной Бритни Спирс; все, что он может вывести, — это определенный ряд последовательно заданных цветных точек-пикселей. Для того чтобы представить информацию в виде, понятном для видеоадаптера, существует мощный графический конвейер. Весь графический конвейер состоит из множества механизмов, некоторые из которых реализуются программно, но основная и большая часть операций происходит аппаратно. Посмотрите на рис. 13.1, где показана общая модель графического конвейера.



Рис. 13.1. Модель графического конвейера

А сейчас для разнообразия давайте займемся "физзарядкой" и "пробежимся" по графическому конвейеру от начала до конца. Весь процесс можно разбить на три условные стадии.

На первой стадии все имеющиеся данные поступают в блок тесселяции, где происходит процесс разбиения поверхности модели на более мелкие части, т. е. полигоны или треугольники. Эту операцию еще называют отбором граней объекта, данная стадия обычно производится программно.

Вторая стадия всегда осуществляется аппаратно, ее обрабатывает графический процессор видеоадаптера. Лет восемь-девять назад этим в основном занимался центральный процессор, и поэтому вся графика целиком была основана на растровых изображениях. Сейчас уровень графических процессоров достаточно мощный, чтобы справиться с хорошей трехмерной графикой. Если вы посмотрите на рис. 13.1, то заметите, что вторая стадия имеет две "тропинки", идя по которым, независимо друг от друга, вы все равно выйдете на главную "дорогу". И вот здесь-то и кроются основные достоинства, которые разберем на данном уроке. Обе "тропинки" или конвейеры, делают одно и то же, а именно, занимаются трансформацией и освещением (Transformation and Lighting). Часто встречается такая аббревиатура записи: T&L. Под трансформацией подразумеваются все мыслимые и немыслимые матричные преобразования, а освещение включает в себя работу с материалом, светом и нормальми. Общая модель работы обоих конвейеров, как говорилось, одна, но вот реализации абсолютно разные.

Третья стадия графического конвейера состоит из нескольких блоков и в целом ее можно назвать стадией растеризации. Если предыдущая стадия производит операции непосредственно с вершинами, то стадия растеризации производит представление объекта на пиксельном уровне. Перед растеризацией происходит отсечение вершин, не попадающих в сцену, и далее в стадии растеризации происходит мультитекстурирование объекта, а также обработка пикселей пиксельными шейдерами. И в конце, прежде чем данные попадают в буфер кадра (frame buffer), происходит ряд так называемых тестов: тест Z-буфера (depth test), тест трафарета (stencil test), альфа-тест (alpha test) и тест смешивания цветов по альфа-каналу (alpha blending). Все тесты происходят на пиксельном уровне, после чего все данные, прошедшие через весь графический конвейер, готовы к выводу на экран.

Рассмотрим особенности фиксированного и программируемого конвейеров.

Фиксированный конвейер

Функционально-фиксированный конвейер (Fixed Function Pipeline) работает традиционным способом. Это конвейер, в котором происходят определенные последовательные шаги по расчету матричных преобразований и расчету освещенности сцены, вне зависимости от того, необходима ли вам какая-

нибудь операция в данный момент или нет. Фиксированный конвейер не дает возможности "прыгать" по конвейеру. Все операции, нужные или не нужные, все равно будут последовательно выполняться. Чтобы, например, рассчитать какой-то источник света и перемножить на необходимую вам матрицу, придется все равно произвести последовательно все операции. То есть фиксированный конвейер лишен определенной гибкости, основан на одних и тех же действиях. Кроме всего прочего, он имеет свои ограничения. Попробуйте включить одновременно десяток источников освещения, и сразу появится необходимость в центральном процессоре. Как этого избежать? На данном этапе решили создать программируемый конвейер.

Программируемый конвейер

Программируемый конвейер (Programmable Pipeline) производит те же самые действия, что и функционально-фиксированный конвейер, по трансформации и освещению объекта, но обработка всех вершин осуществляется при помощи вершинных шейдеров. Фактически произошла замена блока T&L на вершинные шейдеры. Оба конвейера не могут работать одновременно — когда работает один конвейер, другой "спит".

Решать, какой из конвейеров задействовать, вам.

Шейдеры

DirectX 9 имеет два программируемых конвейера — это вершинные шейдеры и пиксельные шейдеры. С помощью вершинных шейдеров до сборки объекта, а точнее вершин, происходят различные матричные преобразования вершин и работа по расчету освещения. Пиксельные шейдеры позволяют производить обработку пикселей цветом, мультитекстурированием, затенением, бликами и многими другими эффектами на пиксельном уровне.

Что вы делаете, когда желаете развернуть объект, осветить его с какой-то из сторон и наложить текстуру? Вы вызываете соответствующие функции из DirectX3D, производя при этом различные установки режимов, определяете необходимость тех или иных флагов, резервируете какое-то количество текстурных уровней, производите всевозможные матричные преобразования — все это очень сильно загружает графический конвейер. Это уже не устраивает разработчиков своим качеством. При увеличении мощности аппаратных средств ситуация не изменится, останется все тот же фиксированный конвейер с массой повторяющихся действий. Добавление, либо улучшение функций DirectX3D, не изменит ситуацию и не внесет видимых изменений, потому что общая модель графического конвейера не изменится. Вот здесь-то и появилось решение в виде добавления программируемого конвейера,

который выполняется графическим процессором видеоадаптера на аппаратном уровне, производя обработку вершин и пикселей, соответственно вершинным и пиксельным шейдерами.

Шейдер — это программа, написанная на языке, подобном ассемблеру, обрабатывающая определенный поток вершинных и пиксельных данных. Шейдеры только обрабатывают данные, они не создают вершины и пиксели. Вы можете произвести, например, матричную конкатенацию с вершиной, но вы не можете создать саму вершину. Шейдеры определяют только обработку данных, создание объекта происходит традиционными способами, изученными на предыдущих уроках.

Вершинные и пиксельные шейдеры обрабатывают за один раз соответственно по одной вершине и по одному пикселу.

В DirectX 9 доступны сразу три версии вершинных и пиксельных шейдеров. Обозначаются они вот таким образом:

▣ vs.1.0, vs.1.1, ps.1.0, ps.1.1, ps.1.3, ps.1.4;

▣ vs.2.0, ps.2.0;

▣ vs.3.0, ps.3.0.

Первая версия вершинных шейдеров была доступна еще с появлением DirectX 8. Эту версию шейдеров поддерживают большинство видеоадаптеров, выпущенных за последние пару лет. В этой версии вершинных шейдеров будут рассматриваться примеры к данному уроку.

Вторая версия вершинных шейдеров претерпела ряд изменений в виде добавления циклов и ветвлений. Эта версия шейдеров в данный момент является минимальной спецификацией, которую обязаны содержать видеоадаптеры с поддержкой DirectX 9.

Третья версия пока доступна на более дорогих видеоадаптерах, но через год-полтора по цене, по всей видимости, они подтянутся к кругу товаров массового потребления. Тогда и стоит о ней вспомнить.

В DirectX 9 вершинные и пиксельные шейдеры реализованы через COM-интерфейсы. Еще в составе DirectX 9 появился высокоуровневый язык программирования шейдеров (High-Level Shader Language) — это очень мощный C-подобный язык программирования шейдеров, с помощью которого можно достигнуть киноэффектов при соответствующей аппаратной поддержке.

По вершинным и пиксельным шейдерам, как мне кажется, уже пора писать отдельную книгу. Они этого достойны как по объему темы, так и по инновации самой технологии. Издательство «БХВ-Петербург» в скором времени планирует выпуск книги по вершинным и пиксельным шейдерам.

Вершинные шейдеры

Вершинный шейдер — это программа, написанная на языке, подобном ассемблеру, с помощью которой возможно произвести геометрические операции над вершинами. Применяя вершинные шейдеры, можно добиться потрясающего качества графики и реализма рисуемой сцены. Имеется возможность мощной анимации, трансформации, деформации объекта и текстур, создания фотореалистичной графики и поверхности любых материалов.

Вершинный шейдер, как уже говорилось, это программа. Такая программа пишется в любом текстовом ASCII-редакторе, например Блокноте, и сохраняется обычно с расширением `vsh` в каталоге вашего приложения. Также имеется возможность написания программы вершинного шейдера в Visual C++ .NET. Откройте любой пустой файл, напишите программу и сохраните с расширением `vsh`. Написание программы вершинного шейдера в Visual C++ .NET дает очевидный плюс, выраженный в виде подсветки синтаксиса. После чего с помощью определенных функций вы подгружаете в ваше приложение программу шейдера.

Архитектура вершинных шейдеров

Прежде чем перейти к синтаксису команд вершинного шейдера, стоит уделить внимание его архитектуре (рис. 13.2).

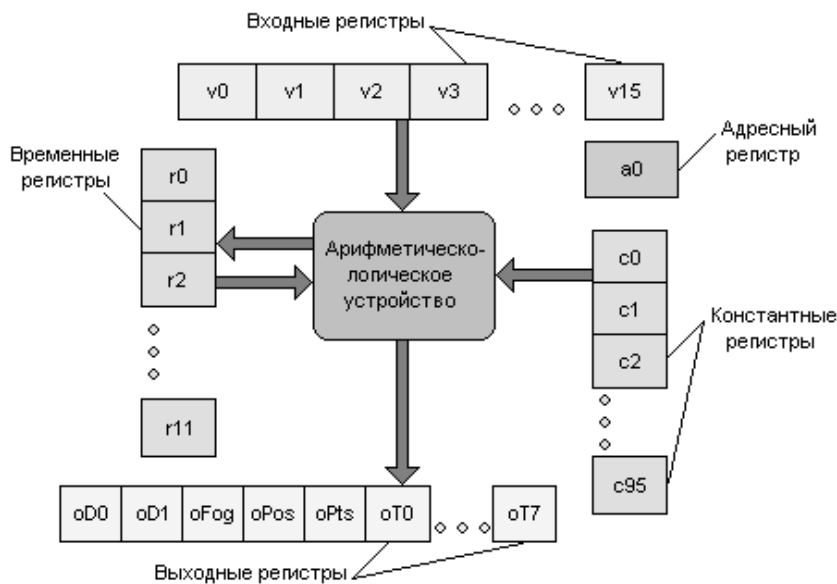


Рис. 13.2. Архитектура вершинного шейдера

Входные регистры (Input registers) обозначены на рис. 13.2 как v_0 — v_{15} , их шестнадцать штук. В эти регистры записывается информация о поступающем потоке данных, который включает в себя информацию о данных вершин. Данные вершин, как мы знаем, содержат координаты самой вершины, текстурные координаты, цвет, нормали и т. д. Каждый регистр, по сути, является вектором, поскольку один регистр содержит четверку чисел с плавающей точкой или `float(x, y, z, w)`.

Константные регистры (Constant registers) помечены как c_0 — c_{95} , таких регистров девяносто шесть штук и они содержат информацию о константах.

Временные регистры (Temporary registers) обозначены как r_0 — r_{11} , всего их насчитывается двенадцать штук, и они содержат некую промежуточную информацию.

Адресный регистр (Address registers) всего один — a_0 . Используется для разного рода адресаций константных регистров.

Выходные регистры (Output registers) — это практически результат той или иной операции, т. е. в эти регистры записываются выходные данные. Их количество не регламентировано и зависит от реализации.

Итак, что же мы имеем? У нас есть определенное количество регистров, с помощью которых имеется возможность производить различные операции над вершиной. Допустим, мы имеем заданную вершину, которую необходимо преобразовать. Доступ к вершине осуществляется через входные регистры. В константных регистрах хранятся матрицы, источники света. Для того чтобы трансформировать объект, необходимо умножить вершину на определенную матрицу, а результат записать либо в выходной регистр, либо во временный. Временный регистр хранит промежуточную информацию, которой можно воспользоваться, для того чтобы, например, перемножить трансформированную вершину на другую матрицу или определенный вектор. Результат помещается в выходной регистр. В этом и заключается вся модель работы шейдера. Но вот чтобы осуществить операции, нужно знать синтаксис команд вершинных шейдеров, необходимых для записи данных в определенные регистры.

Синтаксис команд

Программирование шейдера основывается на массиве команд для каждой из вершин. Такой массив команд можно смело назвать функцией. Максимальное количество команд ограничено числом 128. Для того чтобы произвести какие-либо действия, необходимо соблюдать определенный формат действий:

```
op dest, src0, src1, src2
```

Разберем составляющие данной записи:

□ `op` — это команда заданных действий;

□ `dest` — регистр, в который вы записываете результаты команды;

□ `src(0, 1, 2, ...)` — так обозначаются входные регистры, их количество зависит от команды.

Чтобы воспользоваться форматом записи данных, необходимо изучить синтаксис команд, приведенный в табл. 13.1.

Таблица 13.1. Синтаксис команд

Команда	Назначение	Количество тактов
vs	Версия шейдера	0
add	Сумма	1
dcl_usage	Декларация для входных регистров	0
def	Назначение констант	0
dp3	Скалярное произведение 3D-вектора	1
dp4	Скалярное произведение 4D-вектора	1
dst	Дистанция	1
exp	Экспоненциал	10
expf	Экспоненциал	1
frc	Фрактальный компонент	3
lit	Освещение	1
log	Логарифм	10
logp	Логарифм	1
m3x2	Произведение вектора на матрицу 3x3	2
m3x3	Произведение вектора на матрицу 3x3	3
m3x4	Произведение вектора на матрицу 3x4	4
m4x3	Произведение вектора на матрицу 4x3	3
m4x4	Произведение вектора на матрицу 4x4	4
mad	Произведение и сложение	1
max	Максимум	1
min	Минимум	1
mov	Копирование или запись в регистры	1
mul	Произведение	1
nop	Без операции	0
rcp	Обратное значение	1
rcq	Обратный корень	1

Таблица 13.1 (окончание)

Команда	Назначение	Количество тактов
<code>sge</code>	Если значение больше или равно, то назначить	1
<code>slt</code>	Если значение меньше, то назначить	1
<code>sub</code>	Разность	1

Но это еще не все. Чтобы воспользоваться форматом записи данных и командами для записи, чтения и назначения регистров, необходимо знать, какие из входных и выходных регистров могут содержать определенные данные. Все входные регистры перечислены в табл. 13.2.

Таблица 13.2. Входные регистры

Обозначение	Название	Количество	Операции	Размер
<code>a0</code>	Адрес	1	Записывать и использовать	4D-вектор
<code>c0—c95</code>	Константные	96	Назначать и читать	4D-вектор
<code>r0—r11</code>	Временные	12	Записывать и читать	4D-вектор
<code>v0—v15</code>	Входные	16	Читать	4D-вектор

Всего константных регистров 96 штук. Это самый минимум для вершинных шейдеров версии 1.1 (vs 1.1). Для того чтобы узнать действительное число константных регистров, поддерживаемых вашей видеокартой, нужно воспользоваться структурой `D3DCAPS9` и функцией `MaxVertexShaderConst()`.

Выходные регистры разделены на свои секции и приведены в табл. 13.3.

Таблица 13.3. Выходные регистры

Обозначение	Название	Количество	Операция	Размер
<code>oD0, oD1</code>	Диффузный отражающий свет	2	Запись	4D-вектор
<code>oFog</code>	Коэффициент тумана	1	Запись	Скалярная величина
<code>oPos</code>	Позиция	1	Запись	4D-вектор
<code>oPts</code>	Размер точки	1	Запись	Скалярная величина
<code>oT0—oT7</code>	Координаты текстур	8	Запись	4D-вектор

Понимая синтаксис команд, имея представление о формате записи команд для шейдера, а также зная структуры регистров, можно попробовать написать шейдер:

```
;-----
;   вершинный шейдер
;-----
vs_1_1
dcl_position v0
m4x4 oPos, v0, c0
mov oD0, c4
;-----
```

Разберем составляющие шейдера:

- `vs_1_1` — такая запись указывает на версию используемого шейдера. Это обязательное условие любой программы, и запись инструкции всегда идет первой;
- `dcl_position v0` — с помощью этой инструкции определяется первый входной регистр, в который будет производиться запись;
- `m4x4 oPos, v0, c0` — эти команды осуществляют умножение вектора на матрицу. В программе происходит умножение матрицы, первый элемент которой находится в константном регистре `c0`, на вершину из регистра `v0`. Все полученные от этих действий исходные данные помещаются в выходной регистр `oPos`;
- `mov oD0, c4` — в константном регистре находится цвет вершины. С помощью инструкции `mov` данные помещаются в выходной регистр `oD0` из регистра `c4`.

Это самый простой пример программы вершинного шейдера; надеюсь, вам понятен принцип записи данных внутри программы шейдера. Но сейчас мы подошли к еще одной важной части программирования шейдеров. В примере происходила некая трансформация вершины, но все входные и константные регистры были пустыми. Мы не поместили в эти регистры никаких данных. Для того чтобы заполнить регистры необходимыми данными, в DirectX существуют специальные функции. Допустим, вы пишете код, описывающий создание квадрата, все вершины которого имеют назначенный цвет, после чего производите какие-то операции по матричным преобразованиям и сохраняете, например, полученную матрицу в переменной. После этого заносите данные о вершине во входные регистры и матрицу в константный регистр. И только тогда начинает работать программа шейдера, осуществляя описанные вами действия. Словосочетание "размещаете в регистрах", которое использовалось при описании принципа работы шейдера, очень красочно обрисовывает ситуацию, но оно несколько некорректно и

на самом деле вы только связываете имеющиеся у вас значения с потоком данных, направляя его в необходимые регистры. В DirectX эти действия имеют определение и носят название *декларации*.

Декларация шейдера связывает все необходимые значения при помощи потока данных с входными регистрами шейдера. Для этих целей в DirectX существует типизированный массив данных структуры `D3DVERTEXELEMENT9`, с помощью которой создаются необходимые декларации шейдера. Давайте рассмотрим пример записи декларации, основанной на предлагаемой выше программе шейдера, а потом разберем структуру `D3DVERTEXELEMENT9`, имеющую определенный набор элементов:

```
D3DVERTEXELEMENT9 Declaration[] =
{
    {0, 0, D3DDECLTYPE_FLOAT3,
     D3DDECLMETHOD_DEFAULT,
     D3DDECLUSAGE_POSITION, 0}
    D3DDECL_END()
};
```

Структура `D3DVERTEXELEMENT9` связывает входные данные с конвейером. Прототип структуры выглядит следующим образом:

```
typedef struct _D3DVERTEXELEMENT9 {
    BYTE Stream;
    BYTE Offset;
    BYTE Type;
    BYTE Method;
    BYTE Usage;
    BYTE UsageIndex;
} D3DVERTEXELEMENT9;
```

Структура `D3DVERTEXELEMENT9` имеет такие параметры:

- `Stream` — означает номер потока;
- `Offset` — первый такт для чтения;
- `Type` — тип входных данных;
- `Method` — операторы. Существует несколько операторов, но мы используем по умолчанию `D3DDECLMETHOD_DEFAULT`;
- `Usage` — определяет используемые данные, например, вершины, нормали, текстуры. Имеется множество значений, некоторые из них приведены в табл. 13.4 вместе с соответствующим форматом вершин;
- `UsageIndex` — специфический параметр, пока ставим 0.

Таблица 13.4. Значения используемых данных и форматы вершин

Тип данных	Используемое значение	Формат вершины
D3DDECLTYPE_FLOAT3	D3DDECLUSAGE_POSITION	D3DFVF_XYZ
D3DDECLTYPE_FLOAT4	D3DDECLUSAGE_POSITIONT	D3DFVF_XYZRHW
D3DDECLTYPE_FLOAT3	D3DDECLUSAGE_NORMAL	D3DFVF_NORMAL
D3DDECLTYPE_D3DCOLOR	D3DDECLUSAGE_COLOR	D3DFVF_DIFFUSE
D3DDECLTYPE_FLOATm	D3DDECLUSAGE_TEXCOORD	D3DFVF_SPECULAR
D3DDECLTYPE_FLOAT3	D3DDECLUSAGE_NORMAL	D3DFVF_TEXCOORDS1ZEm(n)

Устанавливая необходимые состояния с помощью данных структуры `D3DVERTEXELEMENT9`, вы определите, как будет происходить связывание вершин с потоком данных. После того как были установлены соответствующие декларации, нужно загрузить вершинные шейдеры из файла. Это делается с помощью трех функций. Функция `IDirect3DDevice9::CreateVertexDeclaration` создает декларацию вершинного шейдера для устройства Direct3D. Функция `D3DXAssembleShaderFromFile()` осуществляет компиляцию шейдера, а функция `CreateVertexShader()` создает вершинный шейдер.

Но перед тем как идти дальше, хотелось бы привлечь ваше внимание к небольшой конструкции кода, проверяющей наличие возможности поддержки графической системы той или иной версии шейдера:

```
D3DCAPS9 Caps;
pDirect3DDevice -> GetDeviceCaps(&Caps);
if (Caps.VertexShaderVersion < D3DVS_VERSION(1,1))
return E_FAIL;
```

При помощи функции `IDirect3DDevice9::GetDeviceCaps` объект структуры `D3DCAPS9` будет содержать в себе параметры установленного видеоадаптера. Макрос `D3DVS_VERSION` производит тест, определяющий возможность работы с необходимой версией шейдера.

А теперь перейдем к коду, загружающему шейдер из файла:

```
LPD3DXBUFFER Code = NULL;
LPDIRECT3DVERTEXSHADER9 pVShader = NULL;
LPDIRECT3DVERTEXDECLARATION9 pVD = NULL;

pDirect3DDevice -> CreateVertexDeclaration(Declaration, &pVD);
D3DXAssembleShaderFromFile("VertexShader.vsh", 0, 0, 0, &Code, 0);
pDirect3DDevice -> CreateVertexShader((DWORD*)
Code -> GetBufferPointer(), &pVShader);
Code -> Release();
```

Рассмотрим прототипы всех используемых функций.

Прототип функции `CreateVertexDeclaration()` описывается так:

```
HRESULT CreateVertexDeclaration(
    CONST D3DVERTEXELEMENT9*    pVertexElements,
    IDirect3DVertexDeclaration9** ppDecl
);
```

Функция `CreateVertexDeclaration()` имеет следующие параметры:

- ❑ `pVertexElements` — переменная, содержащая декларации шейдера. В нашем примере это `Declaration`;
- ❑ `ppDecl` — указатель на интерфейс `IDirect3DVertexDeclaration9`.

Прототип функции `D3DXAssembleShaderFromFile()`:

```
HRESULT WINAPI D3DXAssembleShaderFromFile(
    LPCTSTR          pSrcFile,
    CONST D3DXMACRO* pDefines,
    LPD3DXINCLUDE    pInclude,
    DWORD            Flags,
    LPD3DXBUFFER*    ppShader,
    LPD3DXBUFFER*    ppErrorMsgs
);
```

Функция `D3DXAssembleShaderFromFile()` имеет следующие параметры:

- ❑ `pSrcFile` — имя загружаемого файла, содержащего программу вершинного шейдера;
- ❑ `pDefines` — указатель препроцессора, в книге всегда значение `NULL`;
- ❑ `pInclude` — опциональный указатель интерфейса `ID3DXInclude`, в книге всегда значение `NULL`;
- ❑ `Flags` — флаги, идентифицирующие шейдер. Мы не используем этот параметр, ставим 0;
- ❑ `ppShader` — возвращает буфер, содержащий откомпилированный код шейдера. Здесь используется указатель `Code` на буфер `LPD3DXBUFFER`;
- ❑ `ppErrorMsgs` — сообщение о возврате ошибок, у нас `NULL`.

Прототип функции `CreateVertexShader()` выглядит так:

```
HRESULT CreateVertexShader(
    const DWORD*      pFunction,
    IDirect3DVertexShader9** ppShader
);
```

Функция `CreateVertexShader()` имеет следующие параметры:

- ❑ `pFunction` — указатель, содержащий данные, возвращаемые функцией `ID3DVertexBuffer::GetBufferPointer`. Эта функция не имеет параметров и возвращает указатель на буфер данных. Мы использовали эту функцию на уроке 12;
- ❑ `ppShader` — указатель на интерфейс `IDirect3DVertexShader9`.

После того как шейдер загружен в приложение, а с помощью декларации установлен поток данных, необходимо определить регистры для записи данных. В константные регистры заносятся матрицы преобразований. В примере одна матрица для трансформации. Это итоговая матрица, содержащая преобразования по трансформации вершины. Если предположить, что у нас имеются три матрицы как в предыдущих примерах: матрица вида, мировая матрица и матрица проекции, то формула, содержащая итоговую матрицу, выглядит очень просто:

Итоговая матрица = матрица мировая * матрица вида * матрица проекции

С помощью функции `IDirect3DDevice9::SetVertexShaderConstantF` итоговую матрицу можно занести в необходимый константный регистр:

```
pDirect3DDevice -> SetVertexShaderConstantF(0, (float*)&Matrix, 4);
```

Рассмотрим прототип функции `SetVertexShaderConstantF()`, заносящий данные в регистры:

```
HRESULT SetVertexShaderConstantF(
    UINT          StartRegister,
    CONST float*  pConstantData,
    UINT          Vector4fCount
);
```

Функция `SetVertexShaderConstantF()` имеет следующие параметры:

- ❑ `StartRegister` — первый номер регистра, в который будут записаны данные;
- ❑ `pConstantData` — указатель, содержащий данные для записи в регистр;
- ❑ `Vector4fCount` — количество необходимых регистров для размещения данных.

Поскольку структура регистра — это вектор, состоящий из четырех чисел типа `float`, то матрица 4×4 разместится в четырех регистрах: `c0`, `c1`, `c2`, `c3`.

И последний штрих. Чтобы все произведенные действия вступили в силу, нужно произвести установку вершинного шейдера при помощи функции `IDirect3DDevice9::SetVertexShader`:

```
pDirect3DDevice -> SetVertexShader(pVShader);
```

Рассмотрим прототип функции `SetVertexShader()`:

```
HRESULT SetVertexShader(IDirect3DVertexShader9* pShader);
```

Параметр `pShader` — это указатель на интерфейс `IDirect3DVertexShader9`.

Установить декларацию для вершинного шейдера можно вызовом функции `IDirect3DDevice9::SetVertexDeclaration`, прототип которой очень прост:

```
HRESULT SetVertexDeclaration(IDirect3DVertexDeclaration9 *pDecl);
```

Функция имеет единственный параметр `pDecl` — указатель на интерфейс `IDirect3DVertexDeclaration9`.

Вызов самой функции `SetVertexDeclaration()` будет выглядеть так:

```
pDirect3DDevice -> SetVertexDeclaration(pVD);
```

Мы рассмотрели простой пример использования шейдера, чтобы разобраться в структуре команд шейдера и вызовах соответствующих функций из `Direct3D`. На основе изученного материала все действия по созданию программы шейдера можно разделить на несколько шагов:

1. Создание кода шейдера.
2. Декларирование шейдера.
3. Загрузка шейдера в приложение и компиляция.
4. Связывание данных с входными регистрами шейдера.
5. Установка вершинного шейдера и его декларации.

Рассмотрим более сложный пример вершинного шейдера из документации `DirectX`, используя свет и текстуру.

Структура `CUSTOMVERTEX` должна, естественно, содержать необходимые объявления нормалей, текстурных координат и цвета.

Программа вершинного шейдера может выглядеть следующим образом:

```
;-----
; в регистр v7 заносим цвет
; в регистр v8 заносим текстурные координаты
; в регистре c4 находится матрица
; в регистр c12 заносим освещение
;-----

vs_1_1
dcl_position v0
dcl_normal v4
dcl_color0 v7
dcl_texcoord0 v8
m4x4 oPos, v0, c4
```

```
dp3 r0, v4, c12
mul oD0, r0.x, v7
mov oT0.xy, v8
```

;-----

Рассмотрим составляющие шейдера подробнее:

- ❑ `vs_1_1` — версия шейдера;
- ❑ `dcl_position v0` — первый входной регистр для записи данных;
- ❑ `dcl_normal v4` — входной регистр для нормалей;
- ❑ `dcl_color0 v7` — входной регистр для цвета;
- ❑ `dcl_texcoord0 v8` — входной регистр для текстурных координат;
- ❑ `m4x4 oPos, v0, c4` — произведение вектора на матрицу с последующей записью в выходной регистр;
- ❑ `dp3 r0, v4, c12` — расчет освещения и запись исходных данных в регистр `r0`;
- ❑ `Mul oD0, r0.x, v7` — финальный расчет интенсивности освещения вершины;
- ❑ `Mov oT0.xy, v8` — копирование текстурных координат в выходной регистр.

Декларация вершинного шейдера должна выглядеть соответствующим образом, учитывая использование текстуры, цвета, нормали:

```
D3DVERTEXELEMENT9 Declaration [ ] =
{
    {0, 0, D3DDECLTYPE_FLOAT3,      D3DDECLMETHOD_DEFAULT,
                                     D3DDECLUSAGE_POSITION, 0},
    {0, 12, D3DDECLTYPE_FLOAT3,     D3DDECLMETHOD_DEFAULT,
                                     D3DDECLUSAGE_NORMAL, 0},
    {0, 24, D3DDECLTYPE_D3DCOLOR,   D3DDECLMETHOD_DEFAULT,
                                     D3DDECLUSAGE_COLOR, 0},
    {0, 28, D3DDECLTYPE_FLOAT2,     D3DDECLMETHOD_DEFAULT,
                                     D3DDECLUSAGE_TEXCOORD, 0},

    D3DDECL_END()
};
```

Загрузка файла вершинного шейдера не изменяется, необходимо лишь правильно указать имя самого файла.

Исходя из кода шейдера, запись в константные регистры данных будет следующей:

```
FLOAT Const[4] = {0, 0.5f, 1.0f, 2.0f};
pDirect3DDevice -> SetVertexShaderConstantF(0, (float*)&Const, 1);
```

```
pDirect3DDevice -> SetVertexShaderConstantF(4, (float*)&Matrix, 4);  
FLOAT Color[4] = {1, 1, 1, 1};  
pDirect3DDevice -> SetVertexShaderConstantF(8, (float*)&Color, 1);  
FLOAT DirLighting [4] = {-1, 0, 1, 0};  
pDirect3DDevice -> SetVertexShaderConstantF(12, (float*)&DirLighting, 1);
```

Далее устанавливаются текстурные уровни для текстуры с помощью функций `SetTextureStageState()` и `SetTexture()` по аналогии с действиями, осуществлявшимися на уроке 10.

В функции `RenderingDirect3D()` идентифицируем вершинный шейдер функцией `SetVertexShader()` и буфер вершин функцией `SetStreamSource()`.

На компакт-диске в папке `Urok13` вы найдете пример вершинного шейдера.

Пиксельные шейдеры

Пиксельные шейдеры — это определенная модель обработки пикселей, с помощью которой происходят альфа-смешивание, операции над цветом и адресацией текстур. Попиксельные операции дают возможность достичь максимальной детализации любой поверхности. По аналогии с вершинными шейдерами пишется программа, состоящая из массива команд, и загружающаяся в приложение при помощи функций `Direct3D`. Сама программа создается в любом текстовом редакторе или в `Visual C++ .NET` обычно с расширением файла `psh`.

Архитектура пиксельного шейдера

Архитектура пиксельного шейдера визуально напоминает архитектуру вершинного шейдера, но сильно различается в схеме работы, количестве и назначении регистров. На рис. 13.3 изображена архитектура пиксельного шейдера.

Можно заметить, что константных регистров восемь, входные регистры являются регистрами цвета, а выходной регистр всего один (`r0`).

Формат записи команд обоих шейдеров один и тот же:

```
op, dest, src0, src1...
```

Разберем составляющие данной записи:

- `op` — команда;
- `dest` — регистр;
- `src[0, 1, ...]` — входные регистры.

А вот семантика языка несколько отлична в силу другой модели работы и наличия регистров. Весь процесс работы пиксельных шейдеров разбит на два параллельно существующих конвейера. Один конвейер векторный и отвечает

за работу с значениями RGB, а другой — скалярный конвейер для работы с альфа-каналами. На рис. 13.4 показана модель работы обоих конвейеров.

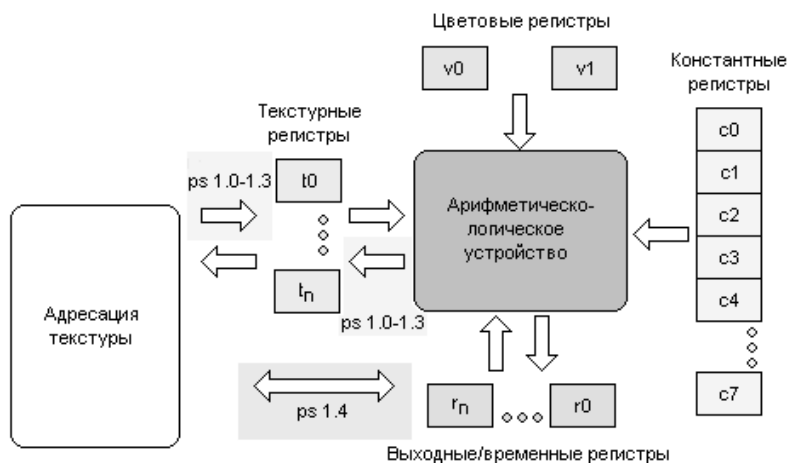


Рис. 13.3. Архитектура пиксельного шейдера

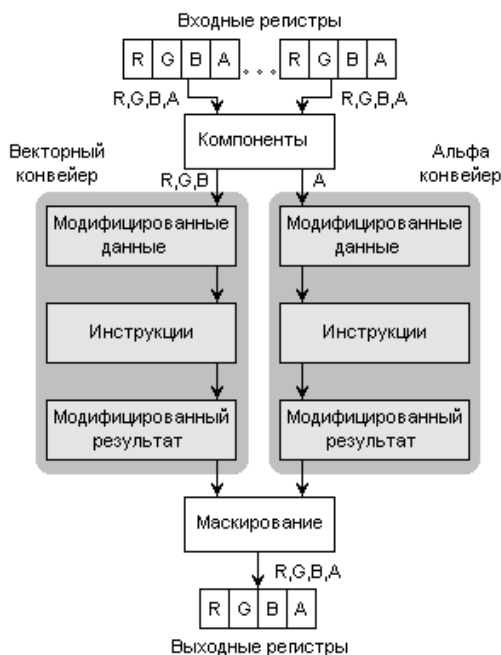


Рис. 13.4. Пиксельный конвейер

Оба конвейера могут выполнять спаренный набор команд, что значительно загрузит процессор и уменьшит общее число машинных циклов пиксельного шейдера.

Синтаксис команд

Прежде чем перейти к синтаксису команд, стоит рассмотреть более подробно типы регистров, которые приведены в табл. 13.5.

Таблица 13.5. Регистры

Обозначение	Имя регистра	Количество	
		Версии 1.0–1.3	Версия 1.4
t0–t4	Текстурные регистры	4	6
rn–r1	Временные регистры	2	6
v0, v1	Цветовые регистры	2	2
c0–c7	Константные регистры	8	8
r0	Выходной регистр	1	1

Каждый регистр выполнен в виде вектора, содержащего значения ARGB. Формат записи известен, теперь перейдем к командам или инструкциям, как их еще иногда называют. Они делятся по значениям на два блока — это текстурные команды и арифметические команды. С помощью арифметических инструкций происходят различные математические операции. Текстурные команды используются для адресации текстур, являясь отдельным набором команд пиксельного шейдера. Текстурные команды всегда находятся в начале программы шейдера. В табл. 13.6 приведены некоторые арифметические команды для версии 1.1, которые мы будем использовать в примере этого урока.

Таблица 13.6. Арифметические команды

Команда	Действие
add	Сумма
cnd	Условный оператор
dp3	Скалярное произведение 3D-вектора
mad	Умножение и сложение
mov	Копирование или запись в регистры
mul	Умножение
sub	Разность

Для версии 1.4 имеется еще ряд определенных команд, информацию о них можно найти в документации DirectX 9 SDK. А теперь рассмотрим текстурные команды, приведенные в табл. 13.7 для версии 1.1.

Таблица 13.7. Текстурные команды

Команды	Действие
<code>tex</code>	Декларация
<code>texbem</code>	Карта рельефного окружения
<code>texbeml</code>	Карта рельефного окружения с освещением
<code>texcoord</code>	Координаты
<code>texm3x2pad</code>	Входные данные умножаются на матрицу 3×2
<code>texm3x2tex</code>	Финальный результат умножается на матрицу 3×2
<code>texm3x3pad</code>	Входные данные умножаются на матрицу 3×3
<code>texm3x3tex</code>	Финальный результат умножается на матрицу 3×3

Пишем пиксельный шейдер

После ознакомления с форматом записи, регистрами и семантикой языка можно попробовать написать программу шейдера. Весь процесс представляет собой ряд последовательных шагов.

1. Проверка поддержки пиксельного шейдера.
2. Установка данных вершин.
3. Написание программы шейдера.
4. Создание и загрузка шейдера.
5. Рендеринг.

Проверка поддержки пиксельного шейдера использует точно такую же запись, как и при проверке вершинных шейдеров, но применяет свой макрос `D3DPS_VERSION(1,1)` для пиксельного шейдера вместе с параметром `PixelShaderVersion` структуры `D3DCAPS9`:

```
D3DCAPS9 Caps;
pDirect3DDevice -> GetDeviceCaps(&Caps);
if (Caps.PixelShaderVersion < D3DPS_VERSION(1,1))
return E_FAIL;
```

Следующий шаг заключается в установке вершинных данных в структуре `CUSTOMVERTEX`. Здесь все естественно зависит от решения поставленной задачи. Возьмем простой пример, где происходит смешивание цвета вершин с

текстурой. Объявление структуры CUSTOMVERTEX и назначение координат цвета, а также текстурных координат может иметь вид:

```
struct CUSTOMVERTEX
{
    FLOAT x, y, z;
    DWORD Color;
    FLOAT tu, tv;
};

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE|
                             D3DFVF_TEX1|D3DFVF_TEXCOORDSIZE2(0))

CUSTOMVERTEX Verhsin[]=
{
    {1.0f, -1.0f, 0.0f, 0xffff00ff, 1.0f, 1.0f},          // A
    {1.0f, 1.0f, 0.0f, 0xff0fff00, 0.0f, 1.0f},          // B
    {-1.0f, 1.0f, 0.0f, 0xffff0000, 0.0f, 0.0f},         // C
    {-1.0f, -1.0f, 0.0f, 0x00000fff, 1.0f, 0.0f},        // D
}
```

Программа шейдера будет простой: наша задача — смешать текстуру с цветом каждой вершины. Значит, необходимо перемножить регистр, содержащий текстурные координаты, с регистром, содержащим цвет вершин, а результат поместить в выходной регистр:

```
;-----
;  смешивание текстуры и цветов
;-----
ps_1_1
tex t0
mul r0, v0, t0
;-----
```

Вот в принципе и весь код. Легко, не правда ли? В первой строке указывается версия используемого пиксельного шейдера, далее происходит размещение текстуры в регистре t0 и в конце осуществляется произведение данных из двух регистров. В регистре v0 находится цвет, итог записывается в выходной регистр r0.

Перед тем как загрузить шейдер, вы должны произвести загрузку необходимой текстуры в приложение при помощи функции D3DXCreateTextureFromFile().

Создание и загрузка пиксельного шейдера происходят в точности как и у вершинного шейдера, но исключение — установка декларации:

```
LPD3DXBUFFER Code = NULL;
LPDIRECT3DVERTEXSHADER9 pPShader = NULL;

D3DXAssembleShaderFromFile("PixelShader.psh", 0, 0, 0, &Code, 0);
pDirect3DDevice -> CreatePexelShader((DWORD*)
Code -> GetBufferPointer(), &pPShader);
Code -> Release();
```

Все абсолютно аналогично, но применяется функция `CreatePixelShader()`, а для создания вершинного шейдера используется однотипная функция `CreateVertexShader()`.

Рендеринг и текстурирование происходят по той же схеме, что и на *уроке 10*, с добавлением функции `SetPixelShader()`, при вызове которой осуществляется установка пиксельного шейдера:

```
pDirect3DDevice -> SetStreamSource(0, pBufferVershin, 0,
                                sizeof(CUSTOMVERTEX));

pDirect3DDevice -> SetFVF(D3DFVF_CUSTOMVERTEX);
pDirect3DDevice -> SetPixelShader(m_pPixelShader);
pDirect3DDevice -> SetTexture(0, pTextura001);
pDirect3DDevice -> DrawPrimitive(D3DPT_TRIANGLELIST, 0, 2);
```

Пиксельные шейдеры позволяют добиться потрясающих эффектов по затенению, бликам закраске и множеству других эффектов, добавляя реализма происходящему на экране монитора.

Итоги урока

Вершинные и пиксельные шейдеры — это мощная технология, обладающая несомненной гибкостью в реализации. Вершинные шейдеры заменяют фиксированный конвейер или модуль T&L в геометрических преобразованиях. Использование пиксельных шейдеров обеспечивает мощную детализацию рисуемой поверхности, благодаря чему достигается возможность киноэффектов в компьютерных играх.

Это был последний урок, посвященный Direct3D. На следующих уроках будут рассматриваться DirectInput, DirectMusic и DirectSound.

Урок 14



Инициализация DirectInput

DirectInput входит в состав библиотеки DirectX и отвечает за работу с устройствами ввода. Так же как и Direct3D, DirectInput обеспечивает аппаратную независимость и высокое быстродействие всех устройств ввода. На рис. 14.1 изображена схема взаимодействия DirectInput8 с аппаратным обеспечением компьютера.



Рис. 14.1. Схема работы DirectInput8

К устройствам ввода относятся:

- ☐ клавиатура;
- ☐ мышь;
- ☐ джойстик, а также любые другие устройства, обеспечивающие взаимодействие приложения и устройства ввода, для получения информации. Например, руль или шлем виртуальной реальности, принадлежащие к устройству типа джойстик.

Режимы данных, получаемых от всех устройств ввода, делятся на два типа: *буферизированные* (buffered) данные и *непосредственные* (immediate) данные.

Буферизированные данные используют виртуальную очередь, сохраняя в ней данные об изменении состояния устройства.

Непосредственные данные возвращают состояния устройства в момент опроса этого устройства. В книге рассматривается режим непосредственного

получения данных. Метод буферизированных данных несколько сложнее в реализации и в книге рассматривается в общем виде.

В свою очередь схема получения данных бывает двух видов: путем оповещения и опроса устройства. Оповещение ожидает получения событий от устройства, и как только происходят какие-либо события, например, нажатие на клавишу, немедленно происходит передача данных. Этот режим в большей степени актуален в многопоточных приложениях. Опрос устройства происходит через заданные промежутки времени.

Интерфейсы

DirectInput в DirectX 9 основан на восьмой версии и базируется, как и вся DirectX, на COM-интерфейсах. Всего имеется три интерфейса:

- IDirectInput8;
- IDirectInputDevice8;
- IDirectInputEffect8.

IDirectInput8 — это самый главный интерфейс, и с его создания начинается любая работа с DirectInput. После создания основного интерфейса происходит инициализация всей библиотеки и получение доступа к двум другим интерфейсам и, как следствие, ко всем структурам, макросам, функциям и т. д.

IDirectInputDevice8 — это второй по значимости интерфейс, создающийся из основного интерфейса. Он отвечает за создание устройств ввода: клавиатуры, мыши и джойстика. Под созданием устройства подразумевается настройка, например, клавиатуры на общее взаимодействие с приложением через интерфейс IDirectInput8. Такая схема взаимодействия показана на рис. 14.2.

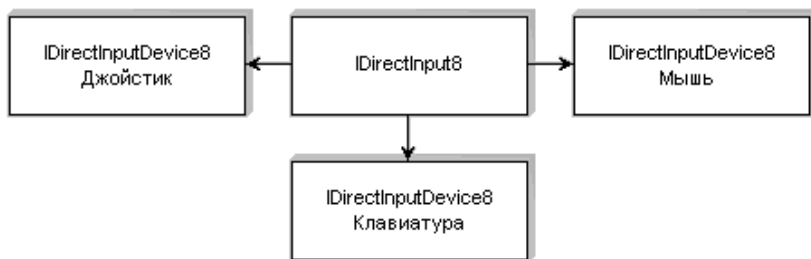


Рис. 14.2. Схема создания устройств

IDirectInputEffect8 — этот интерфейс осуществляет работу с устройствами обратной связи.

Функции **DirectInput8**

DirectInput8, в отличие от **Direct3D**, не содержит такого огромного количества функций, поэтому основную часть этих функций мы сейчас рассмотрим в виде кратких комментариев:

- ❑ **IDirectInput8::CreateDevice** — предназначен для создания устройства **IDirectInputDevice8**;
- ❑ **IDirectInput8::GetDeviceStatus** — дает возможность доступа к устройству **IDirectInput8**;
- ❑ **IDirectInput8::EnumDevice** — идентифицирует определенное устройство, типизируя его сущность;
- ❑ **IDirectInputDevice8::GetDeviceData** — работает с буферизированными данными, извлекая из виртуальной очереди необходимые значения;
- ❑ **IDirectInputDevice8::GetDeviceInfo** — запрашиваемое устройство;
- ❑ **IDirectInputDevice8::GetDeviceState** — используется в режиме непосредственного получения данных, обрабатывая сообщения от устройства;
- ❑ **IDirectInputDevice8::SetCooperativeLevel** — устанавливает уровень кооперации заданного устройства;
- ❑ **IDirectInputDevice8::SetDataFormat** — задает формат получаемых данных от устройства;
- ❑ **IDirectInputDevice8::SetProperty** — позволяет производить настройку ряда параметров устройства;
- ❑ **IDirectInputDevice8::SetEventNotification** — осуществляет обработку оповещений. Применяется в основном в буферизированном режиме;
- ❑ **IDirectInputDevice8::Acquire** — захватывает, восстанавливает потерянное устройство;
- ❑ **IDirectInputDevice8::Unacquire** — уступает доступ к устройству.

Это основная часть функций **DirectInput8**, некоторыми из них мы будем пользоваться.

Создание основного интерфейса

Создание основного интерфейса в **DirectInput8** происходит путем вызова функции **DirectInput8Create()**. Рассмотрим прототип этой функции:

```
HRESULT WINAPI DirectInput8Create(  
    HINSTANCE hinst,  
    DWORD dwVersion,
```

```
REFIID riidIids,
LPVOID* lplpDirectInput,
LPUNKNOWN pUnkOuter);
```

Функция `DirectInput8Create()` имеет следующие параметры:

- ❑ `hinst` — дескриптор экземпляра приложения. Этот дескриптор находится в функции `WinMain()`. В месте, где мы заполняли структуру `WINDCLASSEX` данными, фигурировал параметр `hinstance`, тоже дескриптор экземпляра приложения. Его необходимо сохранить в глобальной переменной или определить глобально и передать функции `DirectInput8Create()`;
- ❑ `dwVersion` — версия `DirectInput`. Это значение обычно `DIRECTINPUT_VERSION`;
- ❑ `riidIids` — версия создаваемого вами интерфейса `IID_DirectInput8`, при желании вы можете обратиться и к более ранней версии;
- ❑ `lplpDirectInput` — адрес переменной, содержащей созданный указатель на COM-интерфейс;
- ❑ `pUnkOuter` — адрес интерфейса наследования `IUnknown` для использования COM-модели. Обычно имеет значение `NULL`.

Чтобы успешно взаимодействовать с `DirectInput`, необходимо подключить соответствующий заголовочный файл `dinput.h`, а также библиотеки `dinput.lib`, `dinput8.lib`, `dxguid.lib`. О подключаемой библиотеке `dxguid.lib` чуть позже на этом уроке. После того как были подключены все заголовочные и библиотечные файлы, можно создать основной объект `DirectInput`, создав прежде указатель на COM-интерфейс `IDirectInput8`:

```
LPDIRECTINPUT8 pInput = NULL;
DirectInput8Create(hwndMain, DIRECTINPUT_VERSION, IID_IDirectInput8,
                  (void**) & pInput, NULL);
```

Обработка ошибок в `DirectInput8` немного отличается от используемой в `Direct3D`. Подробно рассмотрим ее на *уроке 15*. После того как вы создали основной объект интерфейса `IDirectInput8`, вам становятся доступны все остальные интерфейсы, наследуемые из основного. Теперь вы можете переходить к созданию устройства, будь то клавиатура, мышь или джойстик. За создание устройства отвечает интерфейс `IDirectInputDevice8`, наследуемый из интерфейса `IDirectInput8`. С помощью функций, содержащихся в интерфейсе `IDirectInputDevice8`, будет происходить инициализация и настройка всех устройств ввода. Этап по созданию устройств состоит из нескольких шагов. Сейчас будут рассмотрены общие действия по созданию устройств ввода, а на *уроках 15 и 16* — конкретно для клавиатуры и мыши.

Создание устройства ввода

После того как создан основной объект *DirectInput* с помощью функции *DirectInput8Create()*, можно приступить к созданию устройства клавиатуры, мыши или джойстика. Любое создание устройства в *DirectInput8* происходит путем вызова функции *IDirectInput8::CreateDevice*, где одним из параметров является GUID — глобально-уникальный идентификатор. Стандартным устройствам ввода определены свои GUID-идентификаторы:

❑ *GUID_SysKeyboard* — для клавиатуры;

❑ *GUID_SysMouse* — для мыши.

Значение GUID для джойстика и других устройств ввода необходимо определить с помощью функции *IDirectInput8::EnumDevice*, обнаруживающей все устройства, подключенные к компьютеру.

После обнаружения устройства эта функция возвращает два значения: GUID экземпляра и GUID продукта. GUID экземпляра определяет непосредственно одно взятое устройство, например, клавиатуру или мышь, а GUID продукта характеризует общий тип продукта. После чего вызывается функция *IDirectInput8::GetDeviceStatus*, которая решает, есть ли возможность у *IDirectInput8* взаимодействовать с данным типом устройства. Мы не рассматриваем работу джойстика, поэтому будем пользоваться стандартными GUID экземпляра для клавиатуры и мыши. Для этого необходимо подключить библиотеку *dxguid.lib* в ваш проект или использовать директиву *#define INITGUID* в начале файла.

Установка формата данных устройства ввода

После создания устройства необходимо установить формат данных, получаемых от созданного устройства. Все устройства при своем непосредственном использовании формируют определенный пакет данных о событиях, произошедших с ними. Полученный пакет данных должен быть отформатирован соответствующим образом. Формат данных определяется вызовом функции *IDirectInputDevice8::SetDataFormat* с параметром, задающим формат данных для стандартных устройств ввода с помощью таких констант:

❑ *c_dfDIKeyboard* — клавиатура;

❑ *c_dfDIMouse* — мышь;

❑ *c_dfDIJoystick* — джойстик.

Установка уровня взаимодействия устройства ввода

После создания устройства и определения формата получаемых данных осуществляется установка уровня взаимодействия для данного устройства, его еще называют уровнем кооперации. Уровень взаимодействия может быть активным (foreground) или фоновым (background) режимом с эксклюзивным (exclusive) или совместным (noneexclusive) доступом. В активном режиме вы имеете возможность получать данные от устройства в том случае, если оно активно, т. е. находится на переднем плане "в фокусе". В фоновом режиме вне зависимости от "фокуса" приложения доступ к устройству ввода будет постоянным. Эксклюзивный режим позволяет установить соответственно эксклюзивный доступ, где ни одно из приложений не сможет получать доступ к устройству ввода. Тогда как совместный доступ обеспечивает полный доступ всех приложений к созданному устройству. Установка уровня взаимодействия осуществляется вызовом функции `IDirectInputDevice8::SetCooperativeLevel`, где одними из параметров являются флаги, определяющие уровни взаимодействия для создаваемого устройства (они будут рассматриваться на следующих уроках).

Захват устройства ввода

Это очень простой шаг. С помощью функции `IDirectInputDevice8::Acquire` происходит захват устройства для подготовки к получению данных от устройства, а также для восстановления потерянной связи с устройством. Обычно такое происходит, как только приложение утрачивает передний план. Также необходимо упомянуть о функции `IDirectInputDevice8::Unacquire`, которая уступает устройство ввода другим приложениям Windows. Функция `Unacquire()` может использоваться в обработке ошибок при создании устройства клавиатуры, но ее роль несколько больше, чем может показаться на первый взгляд.

Получение данных от устройства ввода

Получение данных от устройства ввода происходит с вызовом функции `IDirectInputDevice8::GetDeviceState`, изымающей информацию от устройства ввода, и помещающей эту информацию в массив данных, из которого происходит их чтение. Параметры этой функции как раз и содержат массив данных и его адрес.

После того как устройство инициализировано, можно получать от него информацию. Принципы обработки сообщений, поступающих от клавиатуры

и мыши, отличаются друг от друга; рассмотрим это на следующих уроках. В книге все примеры основаны на режиме непосредственного получения данных, но о буферизированных данных тоже стоит упомянуть.

Буферизированные данные

Работа с буферизированными данными несколько сложнее, но на начальном этапе настройки устройства почти ничем не отличается от режима непосредственных данных. Для более удобного рассмотрения этой темы разобьем раздел на несколько последовательных шагов:

1. С помощью функции `IDirectInputDevice8::CreateDevice` создается устройство.
2. Установка формата данных для определенного устройства, осуществляется при помощи вызова функции `IDirectInputDevice8::SetDataFormat`.
3. Установка уровня взаимодействия или уровня кооперации обеспечивается функцией `IDirectInputDevice8::SetCooperativeLevel`.
4. С помощью функции `IDirectInputDevice8::SetEventNotification` и структуры `DIPROPDWORD`, содержащей информацию об устройстве, происходит подготовка к буферизации устройства. В конце устройство идентифицируется функцией `IDirectInputDevice8::SetProperty`.
5. Захват устройства происходит с использованием функции `IDirectInputDevice8::Acquire`.
6. Функцией `IDirectInputDevice8::GetDeviceData` осуществляется получение данных от устройства и обработка сообщений.

В целом буферизированный режим очень похож на непосредственный режим получения данных, но имеется ряд ключевых отличий.

Итоги урока

На этом уроке мы рассмотрели *DirectInput8* в целом, пройдя все этапы по инициализации и настройке главного объекта и устройства *DirectInput*. Информация этого урока носила ознакомительно-теоретический характер. На *уроках 15* и *16* будут более подробно рассмотрены этапы создания устройств клавиатуры и мыши.

Урок 15



Работа с клавиатурой

На этом уроке мы рассмотрим стандартные средства для работы с клавиатурой, создадим специальную функцию, в которой инициализируем клавиатуру и настроим ее для работы с пользователем. Весь этап по созданию, инициализации и настройке происходит в несколько шагов:

1. Создание основного объекта `DirectInput`.
2. Создание устройства клавиатуры.
3. Установка формата данных клавиатуры.
4. Установка уровня взаимодействия клавиатуры.
5. Захват доступа к клавиатуре.
6. Получение данных с клавиатуры.
7. Освобождение захваченных ресурсов.

Как вы, наверное, уже заметили, при работе с устройствами DirectX 9 первоочередной задачей является создание этого устройства, но для начала давайте создадим две глобальные переменные, необходимые нам при работе с `DirectInput`:

```
LPDIRECTINPUT8 pInput          = NULL;  
LPDIRECTINPUTDEVICE8 pKeyboard = NULL;
```

Первую переменную `pInput` мы создали на *уроке 14*, она является указателем на основной COM-интерфейс `IDirectInput8`. Вторая переменная — `pKeyboard` — это указатель на интерфейс `IDirectInputDevice8`, представляющий устройство ввода, т. е. клавиатуру.

Создание основного объекта `DirectInput8`

Создайте новый проект, а в нем файл `Keyboard.cpp`, и скопируйте любой понравившийся вам урок из книги. Этот урок не привязан к какому-то конкретному примеру.

Примечание

Не забывайте добавить в ваш проект заголовочный файл `dinput.h` и библиотечные файлы `dinput.lib`, `dinput8.lib`, `dxguid.lib`. А также в самом начале файла `Keyboard.cpp` подключите директиву препроцессора `#define INITGUID`. Иначе при компиляции и компоновке вас неизбежно поджидает ряд ошибок.

Перейдем к созданию новой функции `InitialInput()`, в которую поместим весь код для работы с `DirectInput`.

На предыдущем уроке 14 мы подробно рассмотрели процедуру создания основного объекта `DirectInput` с помощью функции `DirectInput8Create()`, чей вызов в нашей функции и будет первым шагом по созданию основного объекта `DirectInput`:

```
if (FAILED(DirectInput8Create(hinstanceMain, DIRECTINPUT_VERSION,
                             IID_IDirectInput8, (void**)&pKeyboard, NULL)))
    return E_FAIL;
```

Функция `DirectInput8Create()` имеет следующие параметры:

- `hinstance` — дескриптор приложения, передаваемый в функцию `WinMain()`;
- `DIRECTINPUT_VERSION` — константа, указывающая на версию `DirectInput`;
- `IID_IDirectInput8` — версия интерфейса;
- `pInput` — адрес объекта, сохраняющий указатель на интерфейс `IDirectInput8`;
- `NULL` — нулевое значение. Этот параметр не используется.

Создав основной объект `DirectInput8`, можно перейти к созданию устройства клавиатуры.

Создание устройства клавиатуры

Следующим шагом является создание устройства клавиатуры путем вызова функции `IDirectInput8::CreateDevice`. Рассмотрим ее прототип:

```
HRESULT CreateDevice(
    REFGUID          rguid,
    LPDIRECTINPUTDEVICE8 *lplpDirectInputDevice,
    LPUNKNOWN        pUnkOuter)
```

Функция `IDirectInput8::CreateDevice` имеет следующие параметры:

- `rguid` — глобально-уникальный идентификатор GUID создаваемого устройства ввода/вывода. `GUID_SysKeyboard` — определен как глобальный идентификатор клавиатуры;

- ❑ `lplpDirectInputDevice` — адрес переменной, содержащей указатель на интерфейс `IDirectInputDevice8`;
- ❑ `pUnkOuter` — адрес интерфейса наследования `IUnknown` для использования COM-модели. Обычно имеет значение `NULL`.

Создадим устройство клавиатуры:

```
HRESULT hr;  
hr = pKeyboard -> CreateDevice(GUID_SysKeyboard, &pKeyboard, NULL);  
if FAILED(hr)  
{  
    DeleteInput();  
    return FALSE;  
}
```

Обратите внимание на обработку ошибок. Этот код рекомендуется в документации DirectX 9 SDK, в конце урока мы рассмотрим его подробнее. Создав устройство клавиатуры, можно перейти к настройке его свойств.

Установка формата данных клавиатуры

Третий шаг состоит в установке формата данных клавиатуры, необходимого для получения и передачи вводимых данных. Здесь вы просто установите определенный формат данных для клавиатуры посредством вызова функции `IDirectInputDevice8::SetDataFormat`. Ее прототип выглядит следующим образом:

```
HRESULT SetDataFormat(LPCDIDATAFORMAT lpdf);
```

Единственный параметр — `lpdf` — указатель на структуру формата данных, позволяет задавать способы форматирования устройства клавиатуры, мыши, джойстика. Можно воспользоваться такими флагами для определения формата данных по умолчанию: `c_dfDIKeyboard` — для клавиатуры, `c_dfDIMouse` — для мыши.

Установим формат данных для клавиатуры:

```
hr = pKeyboard -> SetDataFormat(&c_dfDIKeyboard);  
if FAILED(hr)  
{  
    DeleteInput();  
    return FALSE;  
}
```

Установка формата данных для клавиатуры выглядит очень просто, но на самом деле это только благодаря использованию функции `SetDataFormat()`, которая делает за нас всю черновую работу.

Установка уровня взаимодействия клавиатуры

Необходимо установить уровень взаимодействия клавиатуры, определяющий доступ вашего приложения к клавиатуре, либо к другому устройству ввода. Установка уровня взаимодействия происходит с помощью функции `IDirectInputDevice8::SetCooperativeLevel`. Ее прототип выглядит таким образом:

```
HRESULT SetCooperativeLevel(  
    HWND hwnd,  
    DWORD dwFlags)
```

Функция `IDirectInputDevice8::SetCooperativeLevel` имеет следующие параметры:

- ❑ `hwnd` — дескриптор окна приложения, находящийся в функции `WinMain()`. Определите эту переменную глобально либо сохраните в глобальной переменной сразу после создания окна в функции `CreateWindowsEx()` из функции `WinMain()`;
- ❑ `dwFlags` — флаги, определяющие уровень взаимодействия с устройством. Рассмотрим некоторые из имеющихся флагов:
 - `DISCL_BACKGROUND` — получает второстепенный доступ;
 - `DISCL_EXCLUSIVE` — получает эксклюзивный доступ. Никакое другое приложение не сможет иметь доступ к устройству;
 - `DISCL_FOREGROVND` — получает доступ переднего плана;
 - `DISCL_NONEXCLUSIVE` — получает неискллючительный доступ;
 - `DISCL_NOWINKEY` — отключает клавишу <ПУСК> в Windows. В исключительном режиме <ПУСК> тоже автоматически отключается.

Все флаги можно комбинировать с помощью побитового ИЛИ "|".

Тогда установка уровня взаимодействия для клавиатуры будет выглядеть следующим образом:

```
hr = pKeyboard -> SetCooperativeLevel(hwndMain,  
                                       DISCL_FOREGROUND|DISCL_NONEXCLUSIVE);  
  
if FAILED(hr)  
{
```

```
DeleteInput();  
return FALSE;  
}
```

Уровень кооперации, выбранный в этом примере, достаточно распространенный и практически является значением по умолчанию, но, как мне кажется, с любым значением всегда интересно поэкспериментировать.

Захват доступа к клавиатуре

Пятый шаг захватывает клавиатуру, это самый простой шаг. Функция, обеспечивающая захват доступа к клавиатуре `IDirectInputDevice8::Acquire`, не имеет параметров:

```
hr = pKeyboard -> Acquire();  
if FAILED(hr)  
{  
    DeleteInput();  
    return FALSE;  
}
```

Осуществив захват устройства можно получать данные, вводимые с клавиатуры.

Получение данных с клавиатуры

Получение данных с клавиатуры осуществляется с помощью функции `IDirectInputDevice8::GetDeviceState`, которая изымает данные с клавиатуры во время запроса. Прототип функции `GetDeviceState()` выглядит следующим образом:

```
HRESULT GetDeviceState(  
    DWORD   cbData,  
    LPVOID  lpvData);
```

Функция `GetDeviceState()` имеет такие параметры:

- ❑ `cbData` — размер массива, в который помещаются данные, изымаемые от устройства. В нашем случае устройство — это клавиатура, поэтому перед вызовом функции `GetDeviceState()` нужно создать массив для хранения данных, получаемых от клавиатуры: `char keyboard[256]`. Клавиатура задается массивом из 256 байтов. Размер массива можно задать оператором `sizeof(keyboard)`;
- ❑ `lpvData` — адрес массива с данными.

Напишем код, для получения данных с клавиатуры:

```
char keyboard[256]
hr = pKeyboard -> GetDeviceState(sizeof(keyboard), (LPVOID) &keyboard);
if FAILED(hr)
{
    DeleteInput();
    return FALSE;
}
```

После вызова функции `GetDeviceState()` все готово, чтобы получать данные. Для этих целей в `DirectInput` имеются специально предопределенные ключи для всего массива данных. Чтобы обработать нажатие клавиши на клавиатуре, используется вот такая запись, проверяющая старший бит:

```
if( keystate[DIK_LEFT] & 0x80 )    // обрабатываем нажатие клавиш
```

Такая запись несколько громоздка, и можно создать макрос, с помощью которого упростится и улучшится запись для обработки нажатий:

```
#define KEYDOWN(name, key) (name[key] & 0x80)
```

Запишите этот макрос в начале рассматриваемого примера и после этого можно описывать код для получения данных с клавиатуры. Например:

```
if (KEYDOWN(keyboard, DIK_LEFT))
{
    // движение объекта в левую сторону
}
if (KEYDOWN(keyboard, DIK_RIGHT))
{
    // движение объекта в правую сторону
}
```

Все пройденные шаги можно суммировать и поместить в одну функцию. В этом примере мы создадим функцию `InitialInput()` и разместим в ней весь изученный код. Но в реальном приложении я рекомендую разделить эту функцию на две отдельные. В первой функции лучше разместить код, инициализирующий клавиатуру. В нем будет происходить создание основного объекта `DirectInput8`, если он не был создан где-либо ранее, создание самой клавиатуры, установка формата данных, уровня взаимодействия клавиатуры и захват клавиатуры. Такую функцию можно поместить в место, где происходят обычные настройки по инициализации различных устройств. Тогда вторая функция содержала бы получение данных от клавиатуры и обработку нажатий клавиш. Такую функцию можно поместить в место, из которого она могла бы влиять на происходящее на экране. В рассматриваемом

примере для простоты все будет находиться в одной функции `InitialInput()`. Посмотрим, как она выглядит:

```
BOOL WINAPI InitialInput()
{
    HRESULT hr;
    char keyboard[256];
    if (FAILED(DirectInput8Create(hinstanceMain, DIRECTINPUT_VERSION,
        IID_IDirectInput8, (void**) &pKeyboard, NULL)))
        return E_FAIL;
    hr = pKeyboard -> CreateDevice(GUID_SysKeyboard, &pKeyboard, NULL);
    if FAILED(hr)
    {
        DeleteInput();
        return FALSE;
    }
    hr = pKeyboard -> SetDataFormat(&c_dfDIKeyboard);
    if FAILED(hr)
    {
        DeleteInput();
        return FALSE;
    }
    hr = pKeyboard -> SetCooperativeLevel(hwndMain,
        DISCL_FOREGROUND|DISCL_NONEXCLUSIVE);
    if FAILED(hr)
    {
        DeleteInput();
        return FALSE;
    }
    hr = pKeyboard -> Acquire();
    if FAILED(hr)
    {
        DeleteInput();
        return FALSE;
    }
    hr = pKeyboard -> GetDeviceState(sizeof(keyboard), (LPVOID) &keyboard);
    if FAILED(hr)
    {
        DeleteInput();
```

```
        return FALSE;
    }
    if (KEYDOWN(keyboard, DIK_RIGHT))
    {
        // обработка правой клавиши <Right> на клавиатуре
    };
    if (KEYDOWN(keyboard, DIK_LEFT))
    {
        // обработка левой клавиши <Left> на клавиатуре
    }
    return TRUE;
}
```

Вызов `InitialInput()` помещается в `WinMain()` перед вызовом функции `RenderingDirect3D()` для того, чтобы непосредственно перед каждым кадром влиять на события, происходящие на экране.

Теперь давайте рассмотрим ключи обработки нажатий клавиш. Ключи, как вы наверно уже догадались, — это константы, определенные в `dinput.h` как структура, отвечающая за устройство клавиатуры. Самих ключей очень много и мы рассмотрим стандартный набор, потому что большое количество ключей отвечают за поддержку языков, основанных на иероглифах, поэтому в книге будет усеченный вариант с полной расшифровкой для каждого ключа. Если у вас возникнет интерес к другим ключам, вы сможете их найти в заголовочном файле `dinput.h`.

Ключи, обрабатывающие нажатие клавиш, выглядят следующим образом:

```
typedef enum {
    DIK_0 - DIK_9,
    DIK_A - DIK_Z,
    DIK_BACK,
    DIK_DELETE,
    DIK_DOWN,
    DIK_END,
    DIK_EQUALS,
    DIK_ESCAPE,
    DIK_F1 - DIK_F15,
    DIK_HOME,
    DIK_INSERT,
    DIK_LCONTROL,
    DIK_LEFT,
```

```
    DIK_LMENU,  
    DIK_LSHIFT,  
    DIK_MINUS,  
    DIK_NEXT,  
    DIK_PAUSE,  
    DIK_RCONTROL,  
    DIK_RETURN,  
    DIK_RIGHT,  
    DIK_RMENU,  
    DIK_RSHIFT,  
    DIK_SPACE,  
    DIK_TAB,  
    DIK_UP,  
} Keyboard Device;
```

Разберем, что обозначает каждая из констант:

- `DIK_0` - `DIK_9` — клавиши от 0 до 9;
- `DIK_A` - `DIK_Z` — клавиши от A до Z;
- `DIK_BACK` — клавиша <Backspace>;
- `DIK_DELETE` — клавиша <Delete>;
- `DIK_DOWN` — клавиша <Page Down>;
- `DIK_END` — клавиша <End>;
- `DIK_EQUALS` — клавиша <=>;
- `DIK_ESCAPE` — клавиша <Esc>;
- `DIK_F1` - `DIK_F15` — клавиши от <F1> до <F15>;
- `DIK_HOME` — клавиша <Home>;
- `DIK_INSERT` — клавиша <Insert>;
- `DIK_LCONTROL` — клавиша <Ctrl> левая;
- `DIK_LEFT` — клавиша <Left>;
- `DIK_LMENU` — клавиша <Alt> левая;
- `DIK_LSHIFT` — клавиша <Shift> левая;
- `DIK_MINUS` — клавиша <_>;
- `DIK_NEXT` — клавиша <Page Down>;
- `DIK_PAUSE` — клавиша <Pause Break>;
- `DIK_RCONTROL` — клавиша <Ctrl> правая;

- ❑ `DIK_RETURN` — клавиша <Enter>;
- ❑ `DIK_RIGHT` — клавиша <Right>;
- ❑ `DIK_RMENU` — клавиша <Alt> правая;
- ❑ `DIK_RSHIFT` — клавиша <Shift> правая;
- ❑ `DIK_SPACE` — клавиша <Space>;
- ❑ `DIK_TAB` — клавиша <Tab>;
- ❑ `DIK_UP` — клавиша <Up>.

Таким образом, задавая любой из ключей, вы сможете определить те или иные действия для каждой из клавиш.

Освобождение захваченных ресурсов

Вернемся к нашему уроку, который уже почти закончился, и нам осталось только рассмотреть функцию `DeleteInput()`, отвечающую за освобождение ресурсов, захваченных `DirectInput`. Она также применяется для обработки ошибок при инициализации и настройке клавиатуры:

```
void WINAPI DeleteInput()
{
    if (pInput)
    {
        if (pKeyboard)
        {
            if (pKeyboard != NULL)
                pKeyboard -> Unacquire();

            if (pKeyboard != NULL)
                pKeyboard -> Release();
            pKeyboard = NULL;
        }
        if (pInput != NULL)
            pInput -> Release();
        pInput = NULL;
    }
}
```

Сама функция абсолютно не сложная. "Двухэтажная" череда операторов `if` приводит нас к обычной процедуре уничтожения указателей на интерфейсы.

Единственное, что для указателя на устройство клавиатуры `pKeyboard` используется вызов функции `IDirectInputDevice8::Unacquire` перед `IDirectInputDevice8::Release`. Это необходимое условие, связанное со спецификой реализации СОМ-модели для `DirectInputDevice8`. Функция `Unacquire()` уступает доступ к устройству `DirectInput`. Функцию `DeleteInput()` поместите в функцию `DeleteDirect3D()`, отвечающую за освобождение всех захваченных ресурсов. Ее название, кстати, можно было бы, и переделать, например, на `DeleteDirectX()`, но это на ваше усмотрение.

Итоги урока

Создав устройство, отвечающее за клавиатуру, вы получаете возможность обрабатывать пользовательские данные. На этом уроке мы прошли через все этапы инициализации и настройки клавиатуры и познакомились с ключами, необходимыми для обработки нажатий клавиш. На *уроке 16* мы рассмотрим другое устройство `DirectInput` — мышь, не менее важное, чем клавиатура.

Урок 16



Мышь

Работа с мышью и джойстиком в DirectInput основана на осевых перемещениях. Любое движение мыши влечет за собой изменение состояния относительно двух осей координат X и Y . Возврат информации о своем состоянии формирует точку местонахождения мыши. Данные, возвращаемые мышью и джойстиком, бывают двух видов — относительные и абсолютные. По умолчанию мышь возвращает относительные данные, а джойстик абсолютные. Но тип данных можно изменять и для мыши, и для джойстика. В работе с мышью мы будем пользоваться относительными данными, т. е. установками по умолчанию. Относительные данные, возникающие при осевых перемещениях мыши, возвращают данные по осям, зависящие от ее последнего положения. Тогда как абсолютные данные возвращают информацию о координатах местонахождения на экране.

Вся работа с мышью сводится к нескольким последовательным шагам, практически идентичным при работе с клавиатурой:

1. Создание основного объекта DirectInput (если он еще не был создан).
2. Создание устройства мыши.
3. Установка формата данных мыши.
4. Установка уровня взаимодействия мыши.
5. Захват доступа к мыши.
6. Получение данных.
7. Освобождение захваченных ресурсов.

Первый шаг в создании основного объекта DirectInput в приложении делается всего один раз и для мыши, и для клавиатуры. Поэтому если вы продолжаете писать код этого урока на основании предыдущего примера, находящегося на компакт-диске в папке Code\Urok15, этот шаг можно пропустить. Если нет, то выглядит это так:

```
if (FAILED(DirectInput8Create(hinstanceMain, DIRECTINPUT_VERSION,  
IID_IDirectInput8, (void**) &pInput, NULL)))  
return E_FAIL;
```

Создание устройства

В данном случае подразумевается, что объект `DirectInput` создан не был. Мы начнем писать код примера к этому уроку с "чистого листа" и первым делом поместим вызов функции `DirectInput8Create()` в функцию `Mouse()`, которая будет отвечать за инициализацию, настройку и захват мыши. Весь код этой функции будет приведен в листинге 16.1.

Создайте новый проект к этому уроку либо модифицируйте предыдущий пример.

Примечание

Если вы создали основной проект, не забывайте подключить к нему заголовочный файл `dinput.h` и библиотечные файлы `dinput.lib`, `dinput8.lib`, `dxguid.lib`. А также в самом начале файла `Keyboard.cpp` подключить директиву препроцессора `#define INITGUID`. Иначе при компиляции и компоновке вас неизбежно поджидает ряд ошибок.

Прежде чем перейти к созданию устройства, необходимо объявить две глобальные переменные:

```
LPDIRECTINPUT8 pInput      = NULL;  
LPDIRECTINPUTDEVICE8 pMouse = NULL;
```

Первая переменная `pInput`, если она не была создана ранее, отвечает за указатель на интерфейс `IDirectInput8`, а вторая переменная `pMouse`, как раз и будет представлять созданное устройство, являющееся в данном примере мышью. Создание устройства мыши выглядит точно так же, как и создание устройства клавиатуры, поэтому прототип функции `CreateDevice()` разбираться не будет. Так же как и все одноименные функции, применяемые на этом уроке. Этот материал подробно рассматривался на *уроке 15*, посвященном работе с клавиатурой. Создание устройства мыши выглядит следующим образом:

```
if (FAILED(pInput -> CreateDevice(GUID_SysMouse, &pMouse, NULL)))  
return E_FAIL;
```

Функция `CreateDevice()` принимает три параметра-значения:

- ❑ `GUID_SysMouse` — это глобально-уникальный идентификатор, определенный для устройства мыши;
- ❑ `pMouse` — указатель на интерфейс `IDirectInputDevice8`, отвечающий за устройство для мыши;
- ❑ `pUnkOuter` — последний параметр не используется и устанавливается в `NULL`.

Установка формата данных

Точно так же, как и для клавиатуры, необходимо установить формат получаемых данных от мыши посредством вызова функции `SetDataFormat()`:

```
if (FAILED(pMouse -> SetDataFormat(&c_dfDIMouse)))
return E_FAIL;
```

Функция `SetDataFormat()` имеет всего один параметр `c_dfDIMouse`, отвечающий за способ форматирования данных, полученных от мыши.

Сам же формат данных для мыши определяется специальной структурой данных `DIMOUSESTATE`. Когда на прошлом уроке 15 вы работали с клавиатурой, то тоже задавали формат данных, но в виде массива из 256 байтов: `char keyboard[256]`. По понятным причинам нельзя задать такой формат данных для мыши, поэтому воспользуемся структурой `DIMOUSESTATE`, прототип которой выглядит таким образом:

```
typedef struct _DIMOUSESTATE
{
    LONG    lX;
    LONG    lY;
    LONG    lZ;
    BYTE    rgbButtons[4];
} DIMOUSE *LPDIMOUSESTATE;
```

Структура DIMOUSESTATE имеет следующие параметры:

- ❑ `lx` — значение по оси X ;
- ❑ `ly` — значение по оси Y ;
- ❑ `lz` — значение по оси Z ;
- ❑ `rgbButtons[4]` — с помощью этого параметра происходит обработка нажатия кнопок мыши.

Установка уровня взаимодействия

Необходимо установить уровень взаимодействия мыши или уровень кооперации с помощью функции `SetCooperativeLevel()`, имеющей два параметра, — это дескриптор окна приложения и флаги, определяющие уровень взаимодействия с устройством:

```
if (FAILED(pMouse -> SetCooperativeLevel(hwndMain, DISCL_FOREGROUND|  
DISCL_EXCLUSIVE)))  
return E_FAIL;
```

Захват доступа к мыши

Захват мыши так же прост, как и захват клавиатуры. Просто вызовите функцию `Acquire()`, не имеющую параметров, и мышь в вашем распоряжении:

```
if (FAILED(pMouse -> Acquire()))  
return E_FAIL;
```

На этом код функции `Mouse()` по инициализации и настройке мыши заканчивается, и вы можете принимать данные от созданного устройства. Функцию `Mouse()` можно вызвать при инициализации всего `DirectX`.

Получение данных

Получение данных от мыши обычно происходит в основном цикле, но прежде необходимо создать глобальную переменную на структуру `DIMOUSESTATE`:

```
DIMOUSESTATE mouse;
```

После чего можно вызвать в основном цикле там, где вам это необходимо, функцию `GetDeviceState()`, при помощи которой изымаются данные, поступающие от мыши. Ее вызов выглядит следующим образом:

```
if (FAILED(pMouse -> GetDevisestate(sizeof(DIMOUSESTATE),  
                                     (LPVOID) &mouse))) ;  
  
return E_FAIL;
```

Затем можно обрабатывать нажатие кнопок мыши. Но так же, как и в случае с клавиатурой, лучше воспользоваться специальными макросами, которые необходимо подключить в начало вашего кода:

```
#define LEFT_BUTTON    0  
#define RIGHT_BUTTON   1  
#define MIDDLE_BUTTON  2
```

Вы, наверное, догадались, что первые две строки отвечают, соответственно, за левую и правую кнопки мыши, а последняя строка — за колесико или кнопку посередине мыши. После чего обработка нажатия кнопок упростится и будет выглядеть следующим образом:

```
if (mouse.rgbButtons[LEFT_BUTTON] & 0x80)  
{  
    // здесь происходит обработка  
    // нажатия левой кнопки мыши  
}
```



```

return E_FAIL;
// захват доступа к мыши
if (FAILED(pMouse -> Acquire()))
return E_FAIL;
return (0);
}

```

Построение класса *MyInput*

Это последний урок, посвященный компоненту DirectInput, поэтому стоит создать класс, отвечающий за инициализацию и настройку DirectInput. Приступая к разработке класса, давайте подумаем, какие из компонентов нам необходимы.

- ❑ Три указателя на интерфейсы. Первый и главный указатель будет отвечать за весь интерфейс IDirectInput8, два других за устройства клавиатуры и мыши.
- ❑ Две переменные. В одной будет содержаться массив данных для клавиатуры, в другой значения для мыши.
- ❑ Функции. На начальном этапе необходимо, по крайней мере, пять функций: инициализация всего DirectInput, создание устройства клавиатуры, создание устройства мыши, освобождение захваченных ресурсов клавиатуры, освобождение захваченных ресурсов мыши.

Будущий класс описан и каких-либо затруднений в создании не вызывает, поэтому напишем спецификацию класса и посмотрим, что получилось:

```

class MyInput
{
private:
char Keyboard[256];
DIMOUSESTATE mouse;
public:
MyInput;
~MyInput();
// указатели на интерфейсы
LPDIRECTINPUT8 pInput;
LPDIRECTINPUTDEVICE8 pKeyboard;
LPDIRECTINPUTDEVICE8 pMouse;
// функции
int CreateInput(HINSTANCE);
int CreateKeyboard(HWND);

```

```
int CreateMouse(HWND);  
void DeleteKeyboard();  
void DeleteMouse();  
}
```

Описание функций класса *MyInput*

Итак, общая модель класса создана, теперь нужно описать каждую из имеющихся функций, к чему мы и приступим. Но прежде опишем конструктор и деструктор:

```
MyInput::MyInput(void)  
{  
    pInput    = NULL;  
    pKeyboard = NULL;  
    pMouse    = NULL;  
};  
  
MyInput::~MyInput(void)  
{  
    pInput    = NULL;  
    pKeyboard = NULL;  
    pMouse    = NULL;  
};
```

В конструкторе и деструкторе происходит обнуление всех имеющихся указателей на интерфейсы. Перейдем к описанию по очереди каждой из функций.

Первая функция — `CreateInput()` — отвечает за создание главного объекта `DirectInput8`. В основе этой функции лежит вызов функции `DirectInputCreate8()`. В качестве параметра в функцию передается контекст приложения `HINSTANCE`, который либо определен глобально, либо сохранен в глобальной переменной:

```
int MyInput::Create(HINSTANCE hinstanceMain)  
{  
    // создание основного объекта  
    if (FAILED(DirectInput8Create(hinstanceMain, DIRECTINPUT_VERSION,  
                                  IID_IDirectInput8, (void**) &pInput, NULL)))  
        return E_FAIL;  
};
```

Следующая функция `CreateKeyboard()` создает и настраивает устройство клавиатуры. В качестве параметра передается дескриптор окна `HWND`. Он,

как и контекст устройства, должен существовать глобально или сохраняться в глобальной переменной.

Вся функция по созданию и настройке клавиатуры основана сразу на нескольких функциях. С помощью этих функций происходят создание клавиатуры, установка формата данных клавиатуры, установка уровня взаимодействия и захват доступа к клавиатуре. Также внутри функции `CreateKeyboard()` происходит вызов функции класса `MyInput::DeleteKeyboard()`:

```
int MyInput::CreateInput(HWND hwndMain)
{
    HRESULT hr;
    // создание устройства клавиатуры
    hr = pInput -> CreateDevice(GUID_SysKeyboard, &pKeyboard, NULL);
    if FAILED(hr)
    {
        DeleteKeyboard();
        return FALSE;
    }
    // установка формата данных клавиатуры
    hr = pKeyboard -> SetDataFormat(&c_dfDIKeyboard);
    if FAILED(hr)
    {
        DeleteKeyboard();
        return FALSE;
    }
    // установка уровня взаимодействия клавиатуры
    hr = pKeyboard -> SetCooperativeLevel(hwndMain,
        DISCL_FOREGROUND|DISCL_NONEXCLUSIVE);
    if FAILED(hr)
    {
        DeleteKeyboard();
        return FALSE;
    }
    // захват доступа к клавиатуре
    hr = pKeyboard -> Acquire();
    if FAILED(hr)
    {
        DeleteKeyboard();
        return FALSE;
    }
};
```

Третья по списку функция — `CreateMouse()`. С ее помощью создается и настраивается мышь. Поскольку мы используем непосредственный режим получения данных от устройства, то реализация этой функции будет подобна функции по созданию клавиатуры:

```
Int MyInput::CreateMouse(HWND hwndMain)
{
    // создание устройства
    if (FAILED(pInput -> CreateDevice(GUID_SysMouse, &pMouse, NULL)))
        return E_FAIL;
    // установка формата данных
    if (FAILED(pMouse -> SetDataFormat(&c_dfDIMouse)))
        return E_FAIL;
    // установка уровня взаимодействия
    if (FAILED(pMouse -> SetCooperativeLevel(hwndMain,
                                              DISCL_FOREGROUND|DISCL_EXCLUSIVE)))
        return E_FAIL;
    // захват доступа к мышши
    if (FAILED(pMouse -> Acquire()))
        return E_FAIL;
};
```

Предпоследняя функция — `DeleteKeyboard()` — будет отвечать за освобождение захваченных ресурсов. Также она использовалась при обработке ошибок, связанных с настройкой клавиатуры. Функция не принимает никаких параметров, поэтому ее тип определен как `VOID`:

```
void MyInput::DeleteKeyboard()
{
    if (pKeyboard != NULL)
        pKeyboard -> Unacquire();

    if (pKeyboard != NULL)
        pKeyboard -> Release();
    pKeyboard = NULL;
};
```

И самая последняя функция создаваемого класса — `DeleteMouse()`, которая освобождает ресурсы, захваченные устройством мыши. Функция также не принимает никаких параметров:

```
void MyInput::DeleteMouse()
{
```

```
if (pMouse != NULL)
    pMouse -> Unacquire();

if (pMouse!= NULL)
    pMouse -> Release();
pMouse = NULL;
};
```

Примечание

Если вы будете использовать макросы для обработки сообщений от клавиатуры и мыши, не забудьте подключить их при помощи директивы препроцессора `#define` в начале заголовочного файла.

Созданный нами класс — это некий "костяк", в который вы по мере необходимости можете добавлять переменные и функции, улучшая функциональность всего класса.

Итоги урока

На уроке были рассмотрены создание и настройка устройства мыши, которые почти ничем не отличались от аналогичных действий по созданию и настройке клавиатуры. При работе с обоими устройствами был использован непосредственный режим получения данных. В конце всего урока, поскольку он последний по `DirectInput`, был создан общий класс, создающий и настраивающий систему `DirectInput`. На *уроке 17* будет рассмотрен компонент `DirectMusic`.

Урок 17



DirectMusic

DirectMusic — это очень большая и сложная система, отвечающая за работу с музыкой в вашем приложении. DirectMusic имеет возможность воспроизводить два формата файлов: WAV и MIDI. На этом уроке будет рассмотрена возможность работы с WAV-файлом.

MIDI (Musical Instrument Digital Interface) — цифровой интерфейс музыкальных инструментов, представляет собой в общем виде поток закодированных данных, состоящий из 16 каналов, каждый из которых отвечает за свой инструмент.

WAV (Waveform Audio) — это формат звуковых данных операционной системы Windows, является стандартом и позволяет кодировать файлы посредством общего шаблона данных.

Интерфейсы DirectMusic

Как и любой компонент из библиотеки DirectX 9 SDK, DirectMusic состоит из ряда интерфейсов, с помощью которых осуществляется работа с COM-объектами. В девятой версии DirectX весь DirectMusic основан на восьмой версии, как и DirectInput. К слову сказать, рассматриваемый на *уроке 18* DirectSound тоже основан на восьмой версии. По всей видимости, это связано с тем, что практически никаких нововведений добавлено не было, поэтому и решили оставить цифру восемь. DirectMusic8 состоит из большого количества интерфейсов, рассмотрим необходимые нам для работы:

- IDirectMusic8;
- IDirectMusicPerformance8;
- IDirectMusicLoader8;
- IDirectMusicSegment8.

IDirectMusic8 — самый главный интерфейс, но его создание не обязательно в силу реализации COM-модели, о которой мы поговорим, когда приступим непосредственно к рассмотрению примера этого урока.

`IDirectMusicPerformance8` — интерфейс, с помощью которого происходит управление музыкальными данными в создаваемом приложении. Также он создает объект `IDirectMusic` и это, опять-таки, связано с COM-моделью.

`IDirectMusicLoader8` — это интерфейс, необходимый для создания загрузчика, при помощи которого вы подгрузите музыкальный файл в ваше приложение.

`IDirectMusicSegment8` — все музыкальные данные загружаются в разделы-сегменты, этот интерфейс отвечает за правильное представление сегментных данных.

Создание приложения

Весь пример этого урока сводится к созданию функции, назовем ее `Music()`, в которой будут происходить инициализация, загрузка и воспроизведение музыки. Поэтому создайте проект `Urok17` и скопируйте в него любой исполняемый файл из пройденных нами уроков, тот, который вам больше нравится. Весь исходный код этого урока вы найдете на прилагаемом к книге компакт-диске в папке `Urok17`.

Компонент `DirectMusic` по своей природе является COM-моделью, и вам необходимо создавать COM-объект с использованием функций, имеющих в COM. Помните, на *уроке 2* подробно разбиралась технология COM, и отмечалось, что нет необходимости "лезть в дебри" COM, что `DirectX 9` представляет для этих целей свои методы решения, но вот `DirectMusic` — как раз исключение из правил. На самом деле хорошо это или плохо — даже не знаю. Все, что нужно сделать, так это обратиться к вызову двух функций из COM: это `CoInitialize()` для инициализации всего COM и `CoCreateInstance()` для создания основного интерфейса `DirectMusic`. Почти то же самое мы делали и в `Direct3D`, и в `DirectInput`. Реализация немного другая, но смысл, в принципе, один и тот же, и никаких трудностей нам это не добавит. Также подключаем библиотеку `dsound.lib`. Она общая для `DirectMusic` и `DirectSound`. Еще необходим заголовочный файл `dmusici.h`, подключите его в начале исполняемого файла:

```
#include <dmusici.h>
```

Приступим к созданию функции `Music()`. Все действия представляют собой ряд последовательных шагов.

1. Инициализация COM.
2. Создание главного интерфейса.
3. Создание загрузчика.
4. Инициализация аудиосистемы.
5. Загрузка файла из каталога.

6. Загрузка сегмента в синтезатор и его воспроизведение.
7. Освобождение захваченных ресурсов.

Инициализация COM

Прежде чем мы пойдем дальше, давайте создадим три указателя на необходимые нам для работы интерфейсы, поместив их в глобальные переменные:

```
IDirectMusicLoader8*    pLoad    = NULL;
IDirectMusicPerformance8* pPerform = NULL;
IDirectMusicSegment8*   pSegment = NULL;
```

Первая переменная `pLoad` отвечает за загрузчик, вторая `pPerform` — за главный интерфейс `DirectMusic` и воспроизведение музыки, и `pSegment` — представляет сегмент данных. Чтобы использовать COM, нужно воспользоваться функцией `CoInitialize()` — это первая и самая главная функция в приложении. При работе с COM необходимо вначале использовать эту функцию для инициализации всего COM, и только потом вы имеете возможность работать с COM-объектами. Сама функция `CoInitialize()` очень простая, принимает всего один параметр. Посмотрим на ее прототип:

```
HRESULT CoInitialize(void* pvReserved);
```

Параметр `pvReserved` задается значением `NULL` и инициализирует COM-библиотеку для данного приложения.

После успешной инициализации можно приступить непосредственно к созданию интерфейса COM-объекта.

Создание главного интерфейса

Создание главного интерфейса происходит вызовом функции `CoCreateInstance()` из COM-библиотеки. Главный интерфейс будет отвечать за всю систему `DirectMusic` в целом, а с помощью указателя `pPerform` появится возможность доступа ко всем функциям интерфейса `IDirectMusicPerformance8`.

Рассмотрим прототип функции `CoCreateInstance()`:

```
HRESULT CoCreateInstance(
    REFCLSID  clsid,
    IUnknown* pUnk,
    DWORD     fdwContext,
    REFIID    iid,
    void**    ppvDest
);
```

Функция `CoCreateInstance()` имеет следующие параметры:

- ❑ `clsid` — в этот параметр передается идентификатор COM-класса для создаваемого объекта. Мы создаем главный интерфейс, значит, нам необходим `CLSID_DirectMusicPerformance`;
- ❑ `pUnk` — содержит указатель на интерфейс `IUnknown` объекта-агрегата. У нас агрегация не используется, поэтому установлено значение `NULL`;
- ❑ `fdwContext` — с помощью этого параметра устанавливается, какие из COM-серверов будут задействованы. Для этого существует три флага:
 - `CLSCTX_INPROC` — реализуется как DLL-библиотека. Мы будем пользоваться этим флагом;
 - `CLSCTX_LOCAL` — реализуется как приложение Windows;
 - `CLSCTX_REMOTE` — автономный сервер;
- ❑ `iid` — в этом параметре нужно указать идентификатор интерфейса для создаваемого экземпляра COM-класса. Для главного интерфейса в `DirectMusic` задан идентификатор `IID_DirectMusicPerformance8`;
- ❑ `ppvDest` — в параметре записывается итог в виде указателя на заданный интерфейс, у нас `pPerform`.

Вот, что получилось в коде:

```
CoCreateInstance(CLSID_DirectMusicPerformance, NULL, CLSCTX_INPROC,  
                IID_IDirectMusicPerformance8, (void**)&pPerform);
```

Так был создан главный интерфейс, отвечающий за весь `DirectMusic` и воспроизведение звуковых файлов.

Создание загрузчика

Сейчас нам необходимо создать загрузчик объектов, с помощью которого будет происходить загрузка музыкального файла. В указателе `pLoad` будет храниться информация о загрузчике объектов, который, в свою очередь, создается с помощью вызова функции `CoCreateInstance()` из библиотеки COM. Прототип этой функции мы уже разобрали, посмотрим, как происходит ее вызов:

```
CoCreateInstance(CLSID_DirectMusicLoader, NULL, CLSCTX_INPROC,  
                IID_IDirectMusicLoader8, (void**)&pLoad);
```

Разберем значения, принимаемые функцией:

- ❑ `CLSID_DirectMusicLoader` — идентификатор COM-класса для созданного объекта;
- ❑ `NULL` — агрегация не используется;

- ❑ `CLSCTX_INPROC` — задействуется COM-сервис, реализуемый при помощи DLL-файлов;
- ❑ `IID_IDirectMusicLoader8` — идентификатор интерфейса создаваемого экземпляра COM-класса;
- ❑ `pLoad` — переменная, содержащая созданный указатель на заданный интерфейс.

Инициализация аудиосистемы

После того как был инициализирован COM, а также с помощью функции `CoCrateInstance()` создан загрузчик и главный интерфейс, отвечающий за всю систему DirectMusic, вам становятся доступны функции из одноименных интерфейсов. Настройку и инициализацию аудиосистемы нужно производить с использованием функции `IDirectMusicPerformance8::InitAudio`. С ее вызовом вы инициализируете всю аудиосистему и настраиваете так называемый аудиопатч. Прототип этой функции не сложный и выглядит таким образом:

```
HRESULT InitAudio(  
    IDirectMusic**      ppDirectMusic,  
    IDirectSound**      ppDirectSound,  
    HWND               hWnd,  
    DWORD               dwDefaultPathType,  
    DWORD               dwPChannelCount,  
    DWORD               dwFlags,  
    DMUS_AUDIOPARAMS*  pParams  
);
```

Функция `IDirectMusicPerformance8::InitAudio` имеет следующие параметры:

- ❑ `ppDirectMusic` — адрес переменной, получающей указатель интерфейса на объект DirectMusic. Здесь мы используем значение `NULL` и тем самым указываем, что будем пользоваться объектом интерфейса `IDirectMusicPerformance8`;
- ❑ `ppDirectSound` — аналогичная ситуация, что и в первом параметре, за исключением того, что все действия связаны с DirectSound. Мы тоже используем здесь значение `NULL`, которое указывает, что DirectMusic создает объект DirectSound и возвращает указатель на этот интерфейс. Вообще, по поводу этих двух параметров не стоит беспокоиться, просто установите значение `NULL` по умолчанию, и DirectMusic внутренне разберется со всем сам. Это также рекомендуется в документации DirectX 9 SDK;

- ❑ `hWnd` — это хендл нашего окна. Сюда можно занести, например, глобальную переменную `hwndMain`, в которой мы сохраняли хендл. Тогда воспроизведение музыки будет происходить, только когда активно все приложение. Если поставить значение `NULL`, то музыка будет проигрываться постоянно, вне зависимости от активности приложения;
- ❑ `dwDefaultPathType` — эта переменная определяет аудиопуть;
- ❑ `dwPChannelCount` — в этой переменной указывается число каналов;
- ❑ `dwFlags` — флаги, с помощью которых устанавливаются эффекты звучания. Всего имеется семь флагов:
 - `DMUS_AUDIOF_3D` — задействован 3D-буфер;
 - `DMUS_AUDIOF_ALL` — используются все установки;
 - `DMUS_AUDIOF_BUFFERS` — используется мультibuфер;
 - `DMUS_AUDIOF_DMOS` — применяются дополнительные возможности;
 - `DMUS_AUDIOF_ENVIRON` — используется окружающий звук;
 - `DMUS_AUDIOF_EAX` — эффекты;
 - `DMUS_AUDIOF_STREAMING` — используется форма поточных волн;
- ❑ `pParams` — адрес структуры `DMUS_AUDIOPARAMS`, задающей параметры синтезатору. Можно установить значение `NULL`, и эта структура не будет требована.

Теперь посмотрим, как выглядит вызов функции `InitAudio()`, и перейдем к загрузке музыкального файла из каталога:

```
pPerform -> InitAudio(NULL, NULL, NULL,
                      DMUS_ APATH_SHARED_STEREOPLUSREVERB,
                      64, DMUS_AUDIOF_ALL, NULL);
```

Загрузка файла из каталога

Итак, мы инициализировали COM, создали главный интерфейс, обеспечили систему загрузчиком, теперь необходимо загрузить музыкальный файл из каталога нашего приложения. Если файл находится не в каталоге приложения, то воспользуйтесь функцией `IDirectMusicLoader8::SetSearchDirectory`. Название загружаемого файла заносится в массив:

```
WCHAR Name[MAX_PATH] = L"Stanislav_Gornakov.wav";
```

В этой конструкции используется литера "L" — это обязательное условие, после которого переменную `Name` можно использовать непосредственно для загрузки. Сам же музыкальный файл `Stanislav_Gornakov.wav` находится в ка-

талог этого урока на прилагаемом компакт-диске. Этот музыкальный файл представляет собой песню, которую я для вас специально написал, вся песня домашнего качества. Певец из меня никудышный, композитор тоже, но придется слушать, а что делать! Кстати, эта ситуация мне напоминает наш шоу-бизнес. Загрузка файла осуществляется с помощью функции `IDirectMusicLoader8::LoadObjectFromFile`. Рассмотрим ее прототип:

```
HRESULT LoadObjectFromFile(
    REFGUID rguidClassID,
    REFIID iidInterfaceID,
    WCHAR* pwzFilePath,
    void** ppObject
);
```

Функция `IDirectMusicLoader8::LoadObjectFromFile` имеет следующие параметры:

- ❑ `rguidClassID` — в этом параметре необходимо указать уникальный идентификатор для объекта, отвечающего за сегмент. Используется идентификатор класса `CLSID_DirectMusicSegment`;
- ❑ `iidInterfaceID` — здесь нужно указать уникальный идентификатор для интерфейса сегмента. Для загрузки в сегмент используется `IID_IDirectMusicSegment8`;
- ❑ `pwzFilePath` — название музыкального файла для воспроизведения. У нас это `Name`;
- ❑ `ppObject` — адрес переменной, получающей указатель на интерфейс `IDirectMusicSegment8`. У нас это `pSegment`.

С параметрами разобрались, посмотрим, как выглядит код загрузки файла из каталога:

```
if (FAILED(pLoad->LoadObjectFromFile(CLSID_DirectMusicSegment,
                                     IID_IDirectMusicSegment8,
                                     Name, (LPVOID*) &pSegment)))
    return E_FAIL;
```

После вызова функции `LoadObjectFromFile()` можно вообще-то использовать небольшую конструкцию для проверки, есть файл в искомом каталоге, и был ли он загружен, вызвав, например, функцию `MessageBox()`. Когда файл успешно загружен, можно перейти к следующему шагу по воспроизведению музыкального файла.

Загрузка сегмента в синтезатор и его воспроизведение

Сегменты в DirectMusic (в дословном переводе это "доли") составляют все звуковые данные. Вне зависимости от формата музыкальный файл должен быть загружен в сегмент. В DirectMusic имеются два типа сегмента: первичный и вторичный. В первичный загружается музыкальный файл, а вторичный отвечает за всевозможные эффекты. Прежде чем воспроизвести файл, необходимо загрузить полученный сегмент в синтезатор с помощью функции `IDirectMusicSegment8::Download`. Очень простая функция, имеет всего один параметр:

```
HRESULT Download(IUnknown* pAudioPath);
```

Параметр `pAudioPath` является указателем на интерфейс `IUnknown`.

Применим функцию `Download()` для загрузки звукового файла:

```
PSegment -> Download(pPerform);
```

Воспользуемся функцией `IDirectMusicPerformance8::PlaySegmentEx`, чтобы воспроизвести файл, загруженный в приложение. Прототип функции описывается так:

```
HRESULT PlaySegmentEx(
    IUnknown*           pSource,
    WCHAR *             pwzSegmentName,
    IUnknown*           pTransition,
    DWORD               dwFlags,
    __int64             i64StartTime,
    IDirectMusicSegmentState** ppSegmentState,
    IUnknown*           pFrom,
    IUnknown*           pAudioPath
);
```

Функция `IDirectMusicPerformance8::PlaySegmentEx` имеет следующие параметры:

- ☐ `pSource` — адрес указателя на сегмент `pSegment`;
- ☐ `pwzSegmentName` — этот параметр в DirectX 9 не используется, поставьте `NULL`;
- ☐ `pTransition` — этот параметр позволяет настраивать переход к сегменту. Может быть установлен в `NULL`;
- ☐ `dwFlags` — параметр, обладающий большим количеством флагов из перечисляемого типа `DMUS_SEGF_FLAGS`. Они позволяют настроить воспроиз-

изведение файла. Но можно выбрать 0, чтобы игнорировать этот параметр;

- ❑ `i64StartTime` — время воспроизведения сегмента. Если поставить 0, воспроизведение произойдет моментально;
- ❑ `ppSegmentState` — позволяет настраивать состояние сегмента, обычно используется значение `NULL`;
- ❑ `pFrom` — с помощью этого параметра можно остановить воспроизведение сегмента в момент начала проигрывания другого сегмента. Можно поставить значение `NULL`;
- ❑ `pAudioPath` — этот параметр указывает на аудиопатч. У нас `NULL`.

В итоге получился следующий вызов функции `PlaySegment()`, состоящий почти из одних нулей:

```
pPerform -> PlaySegmentEx(pSegment, NULL, NULL, 0, 0, NULL, NULL, NULL);
```

Между вызовами функций `Download()` и `PlaySegmentEx()` располагается еще одна функция `IDirectMusicSegment8::SetRepeats`, позволяющая установить количество раз для воспроизведения музыкального файла. Прототип функции такой:

```
HRESULT SetRepeats(DWORD dwRepeats)
```

Единственный параметр `dwRepeats` — отвечает за воспроизведение файла. Можно установить цифры 1, 2, 3 и т. д. Можно установить флаг `DMUS_SEG_REPEAT_INFINITE` для бесконечного повтора. Цифра 0 указывает на единичное проигрывание. Вот что мы запишем:

```
pSegment -> SetRepeats(DMUS_SEG_REPEAT_INFINITE);
```

Теперь у вас на руках есть готовая функция `Music()` и вы можете вызвать ее в любом месте вашего приложения, либо связать ее с какими-то событиями. Это ваше дело. Я же поместил ее в `InitialObject()`, чтобы воспроизведение музыки происходило при загрузке всего приложения. Прежде чем пойти дальше, хотелось бы сказать еще два слова по поводу только что написанной функции `Music()`. Поскольку это был обучающий пример, то инициализацию `DirectMusic` и загрузку с воспроизведением, я поместил в одну функцию. Но лучше поступить более гибко — создать функцию по инициализации `DirectMusic` и поместить ее в `WinMain()`, и отдельно функцию, отвечающую за загрузку и воспроизведение музыкального файла.

Освобождение захваченных ресурсов

После всего, что мы натворили, необходимо "убраться" за собой, освободив захваченные ресурсы с помощью функции `Release()`. Но так как мы ис-

пользовали функции для инициализации COM, то необходимо добавить вызов функции `CoUninitialize()` для деинициализации COM-библиотеки. Как обычно, все вызовы функции, освобождающие захваченные ресурсы, мы помещаем в `DeleteDirectX()`.

Текст функции `Music()` приведен в листинге 17.1.

Листинг 17.1. Функция `Music()`

```
//-----
//  функция
//  Music()
//  воспроизводит музыку в приложении
//-----
HRESULT Music()
{
    //  инициализируем COM
    CoInitialize(NULL);

    //  инициализируем главный интерфейс
    CoCreateInstance(CLSID_DirectMusicPerformance, NULL, CLSCTX_INPROC,
        IID_IDirectMusicPerformance8, (void**)&pPerform);

    //  инициализируем загрузчик
    CoCreateInstance(CLSID_DirectMusicLoader, NULL, CLSCTX_INPROC,
        IID_IDirectMusicLoader8, (void**)&pLoad);

    //  инициализируем аудиосистему
    pPerform -> InitAudio(NULL, NULL, NULL,
        DMUS_APATH_SHARED_STEREOPLUSREVERB,
        64, DMUS_AUDIOF_ALL, NULL);

    //  загружаем музыкальный файл
    WCHAR Name[] = L"Stanislav_Gornakov.wav";
    if (FAILED(pLoad -> LoadObjectFromFile(CLSID_DirectMusicSegment,
        IID_IDirectMusicSegment8,
        Name, (LPVOID*) &pSegment)))

    return E_FAIL;

    //  загружаем сегмент
    pSegment -> Download(pPerform);

    //  повтор
    pSegment -> SetRepeats(DMUS_SEG_REPEAT_INFINITE);
}
```

```
// проигрываем сегмент
pPerform -> PlaySegmentEx(pSegment, NULL, NULL, 0,
                           0, NULL, NULL, NULL);

return 0;
}
```

Итоги урока

На этом уроке вы познакомились с компонентом DirectMusic, отвечающим за музыкальное сопровождение вашего приложения. DirectMusic основан на СОМ-модели, и пришлось воспользоваться СОМ-функциями для создания и инициализации интерфейса. На *уроке 18* рассмотрим DirectSound с помощью классов, предлагаемых библиотекой DirectX.

Урок 18



DirectSound

На этом уроке будет рассмотрен компонент DirectSound. Необходимость этого элемента библиотеки DirectX 9 при разработке игр не так ощутима, как DirectMusic, но он используется, и изучить его структуру стоит.

DirectSound воспроизводит только файлы WAV-формата и строится на основе буферов. Весь смысл работы состоит в создании как минимум одного буфера. Загрузив в этот буфер звуковую информацию, вы можете ее воспроизводить. Вследствие этого все буферы подразделяются на *первичные* и *вторичные*. Первичный буфер (primary buffer) создается автоматически, с созданием объекта DirectSound, и вы не можете запрограммировать этот буфер. Объект DirectSound на самом деле программно представляет звуковую карту, а первичный буфер занимается обработкой имеющихся данных. Вторичный буфер (second buffer) создается программистом и может содержать любую звуковую информацию. Число вторичных буферов ограничено аппаратными средствами компьютера. Вторичные буферы по своим размерам и способу применения делятся на два типа: *статические* и *потокowe*.

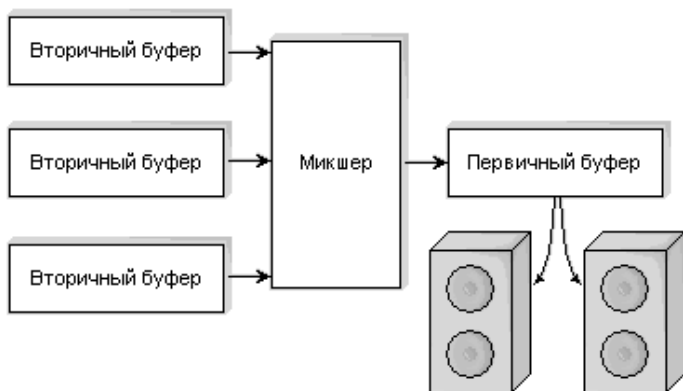


Рис. 18.1. Микширование звука

Статический буфер (static buffer) содержит всю звуковую информацию целиком, поэтому размер такого буфера ограничен. Поточковый буфер (streaming buffer) несколько практичней. Он содержит только часть звуковой информации, а остальные части подгружаются по мере воспроизведения.

После создания нескольких вторичных буферов и помещения в них информации, у вас может возникнуть желание в создании звуковых эффектов путем смешивания звука из двух и более вторичных буферов. Для этой цели в DirectSound имеется *микшер*.

Микшер (mixer), согласно своему названию, позволяет микшировать звуки из вторичных буферов. На рис. 18.1 показана схема такой работы.

Интерфейсы DirectSound

Как и любой другой компонент библиотеки DirectX, DirectSound содержит ряд интерфейсов.

IDirectSound8 — самый главный интерфейс, ассоциируется со звуковым оборудованием, установленным на компьютере. Это самый первый объект, который необходимо создать, после чего вы сможете получить доступ ко всем остальным объектам.

IDirectSoundBuffer8, — как видно из названия, отвечает за буферы. Первичный буфер создается автоматически при создании объекта. Вторичные буферы по мере необходимости создает сам программист.

Так же как DirectInput и DirectMusic, DirectSound базируется на восьмой версии. Существует еще несколько интерфейсов, их описание вы сможете найти в *приложении 1*. Чтобы было легче разобраться в работе DirectSound, разобьем все действия на несколько последовательных шагов.

1. Создание основного объекта.
2. Установка уровня взаимодействия.
3. Создание вторичных буферов.
4. Запись в созданный буфер.
5. Воспроизведение данных.

Необходимо также обеспечить ваше приложение заголовочным файлом dsound.h и библиотекой dsound.lib. Еще одно существенное дополнение — это включение заголовочного файла mmreg.h. Он не принадлежит к DirectX, но будет нужен при описании структуры, содержащей формат звуковых данных. Заголовочный файл mmreg.h принадлежит Win32 API и при подключении в приложение должен находиться перед dsound.h.

Создание основного объекта

Для того чтобы иметь доступ ко всему содержимому, нужно создать основной объект. Создание основного объекта происходит при помощи функции `DirectSoundCreate8()`, что вполне традиционно для DirectX. Прототип функции содержит три параметра и выглядит следующим образом:

```
HRESULT DirectSoundCreate8(
    LPGUID          lpcGuidDevice
    LPDIRECTSOUND*  ppDS8
    IUnknown FAR    pUnkOuter)
```

Разберем параметры функции `DirectSoundCreate8()`:

- `lpcGuidDevice` — содержит GUID звуковой карты. В том случае, когда в системе присутствует одна карта, можно установить `NULL` в качестве значения по умолчанию;
- `ppDS8` — указатель на главный объект `DirectSound`;
- `pUnkOuter` — содержит указатель на интерфейс `IUnknown` объекта-агрегата. Поскольку агрегация не используется, всегда устанавливается значение `NULL`.

Создание основного объекта приобретает такой вид:

```
LPDIRECTSOUND pSound = NULL;
if (FAILED( DirectSoundCreate8(NULL, pSound, NULL)))
    return E_FAIL;
```

Установка уровня взаимодействия

Установка уровня взаимодействия или уровня кооперации происходит с помощью функции `IDirectSound8::SetCooperativeLevel`. Рассмотрим ее прототип:

```
HRESULT SetCooperativeLevel(
    HWND  hwnd,
    DWORD dwLevel)
```

Функция `IDirectSound8::SetCooperativeLevel` имеет следующие параметры:

- `hwnd` — дескриптор окна приложения;
- `dwLevel` — параметр, имеет 4 флага, позволяющих установить уровень взаимодействия. Разберем, что обозначает каждый флаг:
 - `DSSCL_NORMAL` — нормальный стандартный уровень кооперации;
 - `DSSCL_PRIORITY` — приоритетный уровень кооперации;

- `DSSCL_EXCLUSIVE` — эксклюзивный доступ;
- `DSSCL_WRITEPRIMARY` — полный контроль.

Функция принимает два параметра. Первый — это дескриптор приложения, определяющий, какому из приложений задается уровень взаимодействия. Второй параметр задается флагами, с помощью которых происходит установка уровня взаимодействия. Уровень взаимодействия устанавливается для первичного буфера и позволяет либо контролировать буфер, либо не контролировать. Имеется четыре флага для параметра `dwLevel` функции `SetCooperativeLevel()`, а значит, имеется четыре уровня взаимодействия.

- ❑ *Нормальный уровень.* Соответствует флагу `DSSCL_NORMAL`. В этом случае создается первичный буфер, и воспроизведение происходит когда приложение активно.
- ❑ *Приоритетный уровень.* Устанавливается флагом `DSSCL_PRIORITY`. Задавая этот уровень взаимодействия, вы получаете приоритетный доступ к аппаратному обеспечению.
- ❑ *Эксклюзивный уровень.* Задается флагом `DSSCL_EXCLUSIVE`. Идентичен приоритетному, но воспроизведение будет происходить в том случае, когда приложение активно.
- ❑ *Уровень полного контроля устройства.* Этот уровень соответствует флагу `DSSCL_WRITEPRIMARY`. Этот уровень обеспечивает полный контроль, и все настройки и управление первичного буфера должны осуществляться вручную.

Нормальный уровень взаимодействия нам вполне подойдет, это самый, пожалуй, распространенный уровень, им и будем пользоваться. Произведем необходимую настройку уровня кооперации:

```
if (FAILED(pSound -> SetCooperativeLevel(hwndMain, DSSCL_NORMAL)))
return E_FAIL;
```

В `DirectSound8` имеется интерфейс, отвечающий за вторичный аудиобуфер `IDirectSoundBuffer8`. Первое, что нужно, — это создать указатель на данный интерфейс:

```
LPDIRECTSOUNDBUFFER8 pSoundBuffer = NULL;
```

Функция `IDirectSound8::CreateSoundBuffer` создаст один буфер с необходимыми настройками. Рассмотрим прототип этой функции:

```
HRESULT CreateSoundBuffer(
    LPCDSBUFFERDESC    pcDSBufferDesc,
    LPDIRECTSOUNDBUFFER ppDSBuff,
    IUnknown FAR*      pUnkOuter)
```

Функция `IDirectSound8::CreateSoundBuffer` имеет следующие параметры:

- ❑ `pcDSBufferDesc` — член структуры, описывающий создаваемый буфер. Прежде чем вызвать эту функцию, вы должны описать структуру `DSBUFFERDESC`;
- ❑ `ppDSBuff` — указатель на `IDirectSoundBuffer8`;
- ❑ `pUnkOuter` — содержит указатель на интерфейс `IUnknown` объекта-агрегата. Поскольку агрегация не используется, устанавливается значение `NULL`.

Первый параметр этой функции — `pcDSBufferDesc` — член структуры `DSBUFFERDESC`, должен описывать создаваемый буфер. Поэтому, прежде чем воспользоваться функцией `CreateSoundBuffer()`, необходимо создать объект структуры и заполнить соответствующие поля этой структуры. Прототип структуры `DSBUFFERDESC` выглядит так:

```
typedef struct _DSBUFFERDESC{
    DWORD          dwSize;
    DWORD          dwFlags;
    DWORD          dwBufferBytes;
    DWORD          dwReserved;
    LPWAVEFORMATEX lpwfxFormat;
    GUID           guid3DAlgorithm;
} DSBUFFERDESC, *LPDSBUFFERDESC;
```

Структура `DSBUFFERDESC` имеет следующие параметры:

- ❑ `dwSize` — размер создаваемой структуры;
- ❑ `dwFlags` — параметр, имеет флаги, с помощью которых настраивается создаваемый буфер. Рассмотрим некоторые из них:
 - `DSBCAPS_CTRLDEFAULT` — настройки по умолчанию;
 - `DSBCAPS_STATIC` — буфер будет содержать статические данные;
 - `DSBCAPS_LOCSOFTWARE` — буфер будет размещаться в программной памяти;
 - `DSBCAPS_PRIMARYBUFFER` — этот флаг создаст один первичный буфер, и вы обязаны настроить этот буфер вручную;
- ❑ `dwBufferBytes` — размер создаваемого буфера в байтах;
- ❑ `dwReserved` — зарезервированный параметр, должен быть установлен в `NULL`;
- ❑ `lpwfxFormat` — с помощью этого параметра описывается формат содержащихся звуковых данных в создаваемом аудиобуфере. Формат задается с

помощью структуры `WAVEFORMATEX`, и заполнение полей этой структуры должно произойти до описания структуры `DSBUFFERDESC`;

□ `guid3DAlgorithm` — значение по умолчанию `DS3DALG_DEFAULT`.

Предпоследний параметр `lpwfxFormat` структуры `DSBUFFERDESC` является представителем другой структуры `WAVEFORMATEX`, при помощи которой задается формат звука. Рассмотрим структуру `WAVEFORMATEX`, без которой создать вторичный буфер будет трудно:

```
typedef struct _WAVEFORMATEX{
    WORD    wFormatTag;
    WORD    nChannels;
    DWORD   nSamplesPerSec;
    DWORD   nAvgBytesPerSec;
    WORD    nBlockAlign;
    WORD    wBitsPerSample;
    WORD    cbSize;
} WAVEFORMATEX;
```

Структура `WAVEFORMATEX` имеет следующие параметры:

- `wFormatTag` — аудиоформат. Имеет только один флаг `WAVE_FORMAT_PCM`;
- `nChannels` — количество создаваемых каналов;
- `nSamplesPerSec` — частота оцифровки звука;
- `nAvgBytesPerSec` — скорость воспроизводимых данных;
- `nBlockAlign` — блочное выравнивание в байтах;
- `wBitsPerSample` — число бит в сэмпле;
- `cbSize` — дополнительная информация, но чаще всего это значение не используется.

Итак, прежде чем создать вторичный буфер, вы объявляете объект структуры `WAVEFORMATEX`, с помощью которого описывается формат звуковых данных. Далее создается объект структуры `DSBUFFERDESC`, где хранится информация о создаваемом вторичном буфере. В конце следует вызов функции `CreateSoundBuffer()`, непосредственно создающей вторичный буфер. Объединив все вышеописанные действия, получаем работоспособный код, приведенный в листинге 18.1.

Листинг 18.1. Вторичный буфер

```
LPDIRECTSOUND8    pSound          = NULL;
LPDIRECTSOUNDBUFFER pSoundBuffer = NULL;
```

```
DSBUFFERDESC          dsBuffer;
WAVEFORMATEX          wFormat;

// описываем формат звука
ZeroMemory(&wFormat, sizeof(WAVEFORMATEX));
wFormat.wFormatTag     = WAVE_FORMAT_PCM;
wFormat.nChannels      = 2;
wFormat.nSamplesPerSec = 22050;
wFormat.nBlockAlign    = 4;
wFormat.nAvgBytesPerSec = wFormat.nSamplesPerSec * wFormat.nBlockAlign;
wFormat.wBitsPerSample = 16;

// описываем буфер
ZeroMemory(&dsBuffer, sizeof(DSBUFFERDESC));
dsBuffer.dwSize = sizeof(DSBUFFERDESC);
dsBuffer.dwFlags = DSBCAPS_CTRLIPAN | DSBCAPS_STATIC | DSBCAPS_LOCSOFTWARE;
dsBuffer.dwBufferBytes = 3 * wFormat.nAvgBytesPerSec;
dsBuffer.lpwfxFormat = &wFormat;

// создаем буфер
if (FAILED(pSound -> CreateSoundBuffer(&dsBuffer, &pSoundBuffer, NULL)))
return E_FAIL;
```

Запись в созданный буфер

Создав вторичный буфер с соответствующими настройками, можно произвести запись аудиоданных в буфер. Запись данных происходит подобно записи данных вершин в буфер вершин. Вначале вторичный буфер блокируется для записи, необходимые данные копируются в буфер, и после этого буфер разблокируется.

Заблокировать буфер можно с помощью функции `IDirectSoundBuffer8::Lock`, которая имеет семь параметров, в отличие от своей "тезки" из интерфейса `IDirect3DVertexBuffer`. В буфере хранятся аудиоданные, считываемые при помощи курсора воспроизведения, а записанные при помощи курсора записи. Оба курсора — это метки, при помощи которых помечаются данные для воспроизведения и записи. Такой способ работы вызван ситуацией, в которой при воспроизведении звука вы можете производить запись в буфер. В том случае, если выставить метки в буфере, то будет известно, где нахо-

дятся данные для воспроизведения и свободное место в буфере для записи. Рассмотрим прототип функции `Lock()` для блокировки вторичного буфера:

```
HRESULT Lock(  
    DWORD    dwOffset,  
    DWORD    dwBytes,  
    LPVOID*  ppvAudioPtr1,  
    LPDWORD  pdwAudioBytes1,  
    LPVOID*  ppvAudioPtr2,  
    LPDWORD  pdwAudioBytes2,  
    DWORD    dwFlags  
);
```

Функция `Lock()` имеет следующие параметры:

- `dwOffset` — расположение курсора записи;
- `dwBytes` — размер блокируемого буфера в байтах;
- `ppvAudioPtr1` — адрес указателя на массив первой части данных;
- `pdwAudioBytes1` — адрес размера массива первой части данных;
- `ppvAudioPtr2` — адрес указателя на массив второй части данных;
- `pdwAudioBytes2` — адрес размера массива второй части данных;
- `dwFlags` — параметр, имеет два флага, определяющие способ блокировки буфера. Рассмотрим их действие:
 - `DSBLOCK_ENTIREBUFFER` — блокируется весь буфер;
 - `DSBLOCK_FROMWRITECURSOR` — блокируется только часть буфера исходя из позиции курсора записи.

В функции `Lock()` имеются два указателя на две части массива данных буфера. Это те самые метки, о которых мы говорили, перед тем как разобрать функцию `Lock()`. Так вот, первый указатель должен находиться в начале данных, записанных в буфер, а второй — в начале оставшейся пустой области буфера. Причем размер второй части массива данных не должен быть больше фактического размера оставшейся части свободного буфера, иначе произойдет ошибка. Два этих указателя и являются курсорами записи и воспроизведения.

После блокировки буфера можно произвести запись данных в буфер. Запись данных в буфер я умышленно опускаю вследствие того, что функций, которые загружали бы в приложение аудиофайл и делали запись в буфер, в `DirectSound` нет. Вернее, они есть, но доступны из классов `CSoundManager`, `CSound`, `CWaveFile` и `CStreamingSound`. Чтобы не отвлекаться, рассмотрим до конца весь процесс работы, вплоть до воспроизведения данных, а потом разберем указанные выше классы, которые намного облегчат общение с `DirectSound`. К слову сказать, все то, что мы сейчас изучаем, описано в этих

классах с помощью своих функций, и проще пользоваться объектами этих классов, нежели писать свои функции. Но общая структура работы с этими классами построена на действиях, сейчас изучаемых нами, поэтому данный материал пригодится.

Разблокирование буфера происходит при использовании функции `IDirectSoundBuffer8::Unlock`. Рассмотрим ее прототип:

```
HRESULT Unlock(  
    LPVOID pvAudioPtr1,  
    DWORD  dwAudioBytes1,  
    LPVOID pvAudioPtr2,  
    DWORD  dwAudioBytes2  
);
```

Функция `IDirectSoundBuffer8::Unlock` имеет следующие параметры:

- ❑ `pvAudioPtr1` — адрес указателя на массив первой части данных;
- ❑ `dwAudioBytes1` — размер первой части в байтах;
- ❑ `pvAudioPtr2` — адрес указателя на массив второй части данных;
- ❑ `dwAudioBytes2` — размер второй части в байтах.

Воспроизведение данных

Поместив аудиоданные в буфер, вы можете воспроизвести их с помощью функции `IDirectSoundBuffer8::Play`. Рассмотрим ее прототип:

```
HRESULT Play(  
    DWORD  dwReserved1,  
    DWORD  dwPriority,  
    DWORD  dwFlags)
```

Функция `IDirectSoundBuffer8::Play` имеет следующие параметры:

- ❑ `dwReserved1` — зарезервированный параметр, должен иметь значение 0;
- ❑ `dwPriority` — установка приоритета воспроизведения. По умолчанию имеет нулевое значение;
- ❑ `dwFlags` — параметр определяет схему воспроизведения. Имеет флаг `DSBPLAY_LOOPING`, зацикливающий воспроизведение. Также можно установить целочисленное значение, указывающее число повторений воспроизведения. Ноль соответствует одному разу.

Чтобы остановить воспроизведение данных, можно воспользоваться функцией `IDirectSoundBuffer8::Stop`, которая не имеет абсолютно никаких параметров, но "коня на скаку остановит".

Классы

А теперь обещанные классы, с помощью которых вы избавитесь от множества головных болей и которыми лично я пользуюсь постоянно. Имеется четыре класса, реализующих различные способы работы с DirectSound:

- ❑ `CSound` — представляет статический буфер, содержащий всю звуковую информацию целиком;
- ❑ `CSoundManager` — инициализирует DirectSound, создавая первичный и вторичный буферы;
- ❑ `CStreamingSound` — представляет потоковый буфер, содержащий звуковые данные по частям;
- ❑ `CWaveFile` — используется для чтения и записи WAV-файлов из различных ресурсов.

Все четыре класса находятся в файлах `dsutil.h` и `dsutil.cpp`, где даны соответственно спецификация классов и их реализация. Также необходимо подключить библиотеку `winmm.lib`. Разберем вкратце каждый из классов.

Класс *CSound*

Этот класс работает с одним или несколькими статическими буферами. Спецификация класса `CSound` приведена в листинге 18.2.

Листинг 18.2. Класс `CSound`

```
class CSound
{
    protected:
        LPDIRECTSOUNDBUFFER* m_apDSBuffer;
        DWORD m_dwDSBufferSize;
        CWaveFile* m_pWaveFile;
        DWORD m_dwNumBuffers;
        DWORD m_dwCreationFlags;

        HRESULT RestoreBuffer(LPDIRECTSOUNDBUFFER pDSB, BOOL* pbWasRestored);

    public:
        CSound(LPDIRECTSOUNDBUFFER* apDSBuffer,
               DWORD dwDSBufferSize, DWORD dwNumBuffers,
               CWaveFile* pWaveFile,
```

```
        DWORD dwCreationFlags);  
virtual ~CSound();  
  
HRESULT Get3DBufferInterface(DWORD dwIndex,  
                             LPDIRECTSOUND3DBUFFER* ppDS3DBuffer);  
HRESULT FillBufferWithSound(LPDIRECTSOUNDBUFFER pDSB,  
                             BOOL bRepeatWavIfBufferLarger);  
LPDIRECTSOUNDBUFFER GetFreeBuffer();  
LPDIRECTSOUNDBUFFER GetBuffer(DWORD dwIndex);  
HRESULT Play(DWORD dwPriority = 0, DWORD dwFlags = 0,  
             LONG lVolume = 0, LONG lFrequency = 0);  
HRESULT Play3D(LPDS3DBUFFER p3DBuffer, DWORD dwPriority = 0,  
              DWORD dwFlags = 0, LONG lFrequency = 0);  
HRESULT Stop();  
HRESULT Reset();  
BOOL IsSoundPlaying();  
};
```

Конструктор класса `CSound` заполняет буферы данными из указанного WAV-файла и инициализирует их необходимыми значениями.

Рассмотрим функции класса `CSound`:

- ❑ `FillBufferWithSound()` — заполняет буфер аудиоданными из WAV-файла;
- ❑ `Get3DBufferInterface()` — если буфер имеет трехмерные возможности, то извлекается указатель `IDirectSoundBuffer8` на необходимый буфер;
- ❑ `GetBuffer()` — извлекает необходимый интерфейс `IDirectSoundBuffer`. Чтобы получить нужный интерфейс, воспользуйтесь функцией `QueryInterface`, рассмотренной на *уроке 2*;
- ❑ `GetFreeBuffer()` — извлекает интерфейс `IDirectSoundBuffer` для первого незадействованного буфера;
- ❑ `IsSoundPlaying()` — возвращает булеву переменную, которая показывает, какой из буферов воспроизводится;
- ❑ `Play()` — воспроизводит буфер;
- ❑ `Play3D()` — воспроизводит 3D-буфер;
- ❑ `Reset()` — устанавливает курсор воспроизведения на начало во всех имеющихся буферах. С помощью этой функции происходит сброс курсоров;
- ❑ `Stop()` — останавливает воспроизведение.

Не сложный и интуитивно понятный класс, экономящий массу времени. Перейдем к следующему классу.

Класс *CSoundManager*

Класс `CSoundManager` инициализирует `DirectSound`, первичный буфер и создает заданное количество вторичных буферов. Объявление класса приведено в листинге 18.3.

Листинг 18.3. Класс `CSoundManager`

```
class CSoundManager
{
    protected:
        LPDIRECTSOUND8 m_pDS;
    public:
        CSoundManager();
        ~CSoundManager();
        HRESULT Initialize(HWND hWnd, DWORD dwCoopLevel);
        inline LPDIRECTSOUND8 GetDirectSound() {return m_pDS;}
        HRESULT SetPrimaryBufferFormat(DWORD dwPrimaryChannels,
                                       DWORD dwPrimaryFreq,
                                       DWORD dwPrimaryBitRate);
        HRESULT Get3DListenerInterface(LPDIRECTSOUND3DLISTENER*
                                       ppDSListener);
        HRESULT Create(CSound** ppSound, LPTSTR strWaveFileName,
                      DWORD dwCreationFlags = 0,
                      GUID guid3DAlgorithm = GUID_NULL,
                      DWORD dwNumBuffers = 1);
        HRESULT CreateFromMemory(CSound** ppSound, BYTE* pbData,
                                ULONG ulDataSize, LPWAVEFORMATEX pwfx,
                                DWORD dwCreationFlags = 0,
                                GUID guid3DAlgorithm = GUID_NULL,
                                DWORD dwNumBuffers = 1);
        HRESULT CreateStreaming(CStreamingSound** ppStreamingSound,
                                LPTSTR strWaveFileName,
                                DWORD dwCreationFlags,
                                GUID guid3DAlgorithm,
                                DWORD dwNotifyCount,
```

```
        DWORD dwNotifySize,
        HANDLE hNotifyEvent);
```

```
};
```

Конструктор инициализирует данные класса.

Рассмотрим функции класса `CSoundManager`:

- ❑ `Create()` — создает объект `Csound`, ассоциирующийся с WAV-файлом;
- ❑ `CreateFromMemory()` — создает объект `Csound`;
- ❑ `CreateStreaming()` — создает объект `CstreamingSound`, ассоциирующийся с WAV-файлом и потоковым буфером;
- ❑ `Get3DListenerInterface()` — применяется для извлечения интерфейса `IDirectSound3DListener8`;
- ❑ `GetDirectSound()` — извлекает интерфейс `IDirectSound8` объекта `DirectSound`, созданного функцией `Initialize()`;
- ❑ `Initialize()` — создает объект `DirectSound` и устанавливает уровень взаимодействия;
- ❑ `SetPrimaryBufferFormat()` — устанавливает формат первичного буфера.

Также не сложный класс, производящий почти все первичные настройки и инициализацию.

Класс *CStreamingSound*

Класс `CStreamingSound` представляет потоковый буфер и наследуется из `CSound`. В листинге 18.4 приведен текст рассматриваемого класса.

Листинг 18.4. Класс *CStreamingSound*

```
class CStreamingSound : public CSound
{
protected:
    DWORD m_dwLastPlayPos;
    DWORD m_dwPlayProgress;
    DWORD m_dwNotifySize;
    DWORD m_dwNextWriteOffset;
    BOOL m_bFillNextNotificationWithSilence;
public:
    CStreamingSound(LPDIRECTSOUNDBUFFER pDSBuffer, DWORD
dwDSBufferSize,
    CWaveFile* pWaveFile, DWORD dwNotifySize);
```

```

~CStreamingSound();

HRESULT HandleWaveStreamNotification(BOOL bLoopedPlay);

HRESULT Reset();

};

```

Конструктор класса `CStreamingSound` вызывает конструктор класса `CSound` для заполнения буфера данными и инициализации имеющихся переменных.

Рассмотрим две функции класса `CStreamingSound`:

- ❑ `HandleWaveStreamNotification()` — восстанавливает буфер, если он был потерян, читает данные из файла и записывает их в часть буфера, которая должна быть воспроизведена;
- ❑ `Reset()` — восстанавливает буфер и сбрасывает курсор воспроизведения в начало.

Класс *CWaveFile*

Этот класс необходим для чтения, записи и загрузки WAV-файлов. Объект этого класса создается `CsoundManager` при создании объекта `CSound` или `CStreamingSound`. Спецификация класса приведена в листинге 18.5.

Листинг 18.5. Класс *CWaveFile*

```

class CWaveFile
{
public:
    WAVEFORMATEX* m_pwfx;
    HMMIO          m_hmmio;
    MMCKINFO       m_ck;
    MMCKINFO       m_ckRiff;
    DWORD          m_dwSize;
    MMIOINFO       m_mmioinfoOut;
    DWORD          m_dwFlags;
    BOOL           m_bIsReadingFromMemory;
    BYTE*          m_pbData;
    BYTE*          m_pbDataCur;
    ULONG          m_ulDataSize;
    CHAR*          m_pResourceBuffer;
protected:
    HRESULT ReadMMIO();
    HRESULT WriteMMIO(WAVEFORMATEX *pwfxDest);
};

```

```

public:
    CWaveFile();
    ~CWaveFile();

    HRESULT Open(LPTSTR strFileName, WAVEFORMATEX* pWfx, DWORD dwFlags);
    HRESULT OpenFromMemory(BYTE* pbData, ULONG ulDataSize,
                           WAVEFORMATEX* pWfx, DWORD dwFlags);

    HRESULT Close();

    HRESULT Read(BYTE* pBuffer, DWORD dwSizeToRead, DWORD* pdwSizeRead);
    HRESULT Write(UINT nSizeToWrite, BYTE* pbData, UINT* pnSizeWrote);
    DWORD GetSize();
    HRESULT ResetFile();
    WAVEFORMATEX* GetFormat() {return m_pWfx;};
};

```

Конструктор инициализирует все переменные.

Рассмотрим функции класса `CWaveFile`:

- ❑ `Close()` — закрывает файл;
- ❑ `GetFormat()` — возвращает объект структуры `WAVEFORMATEX`, описывающий формат звука;
- ❑ `GetSize()` — извлекает размер данных после их открытия;
- ❑ `Open()` — очень мощная функция, с ее помощью можно открыть, записать, загрузить, извлечь WAV-файл;
- ❑ `OpenFromMemory()` — инициализирует данные с последующим вызовом для чтения из памяти;
- ❑ `Read()` — читает данные из файла, ресурса или памяти и копирует в буфер, заблокированный функцией `IDirectSoundBuffer8::Lock`;
- ❑ `Write()` — записывает данные в файл из адреса.

С помощью всех четырех рассмотренных классов очень легко работать с `DirectSound`. Можно комбинировать свои функции с объектами и функциями рассмотренных классов, а можно пользоваться только объектами этих классов. На прилагаемом к книге компакт-диске в папке `Urok18` находится пример к этому уроку.

Итоги урока

`DirectSound` не столь востребованный интерфейс, как `DirectMusic`, но он обладает мощными средствами для работы с аудиоданными, а четыре рассмотренных класса облегчают поставленную задачу.

Урок 19



Итоги

Это последний урок в этой книге и он посвящен компоновке приложения.

Язык программирования C++ — объектно-ориентированный, и в этом заключаются все его лучшие качества. Что касается всех изученных уроков, то, как уже говорилось, столь маленькие примеры не имело смысла разбивать на классы. Объектно-ориентированная компоновка в конкретном случае только запутала бы ситуацию, нежели внесла гибкости в разрабатываемое приложение.

Объектно-ориентированное программирование, в самом деле, гениальная идея и, что самое главное, доступная в освоении. Я думаю, что модель создания классов вам известна, любая книга по C++ обязательно содержит по этой теме исчерпывающие сведения, поэтому каких-либо затруднений в создании классов возникнуть не должно. Давайте сейчас схематично рассмотрим один из сотни всевозможных способов структуры создания приложения, а потом я вам предложу более заманчивый вариант развития дальнейших событий.

Создание модели классов для какого-то конкретного приложения, по моему, может ограничиваться лишь вашей фантазией. Способов реализации действительно много.

В течение всей книги было изучено несколько главных компонентов DirectX 9 — это Direct3D, DirectMusic, DirectInput, DirectSound, поэтому создание классов для каждого компонента выглядит вполне логичной идеей. С другой стороны, никто вам не мешает создать класс, например, отвечающий за загрузку мэша, или класс мыши, который будет дружественным к общему классу DirectInput. В принципе, всем действиям по созданию того или иного класса, его внутренним функциям, переменным, конструктору, деструктору, можно попытаться дать вектор направления в плане общей модели построения класса. Посмотрите на рис. 19.1, где показана общая идея формирования какого-то N-класса.



Рис. 19.1. Схема класса

Эта схема очень простая. Конструктор и деструктор служат соответственно для создания и удаления объектов класса. Члены класса — это данные класса, а функции предназначены для взаимодействия с этими данными. Имея представление о модели класса и о синтаксисе языка программирования C++, можно создавать и развивать любое приложение. Конечно, более детальная разработка определенного класса это первостепенная задача, но все же это несколько не трудное занятие, а по мне, так даже более интересное, чем набивка всего кода в целом.

А сейчас о том самом заманчивом варианте, предложенном вам в начале урока. Он очень хорошо поможет нам разобраться в модели компоновки классов, и, что самое главное, познакомит с массой новых интересных деталей. На *уроке 2* после установки DirectX 9 SDK было создано демонстрационное приложение, производящее инициализацию и настройку DirectX 9. В созданном примере на экране рисовался красный чайник, который можно было развернуть в любую сторону. Это демонстрационное приложение на самом деле представляет очень мощный набор классов и функций, избавляющий вас как от массы написания кода, повторяющегося из приложения в приложение, так и от создания основной структуры объектно-ориентированного приложения. С помощью этого мастера создается ряд хорошо продуманных классов, отвечающих за обработку, инициализацию и создание всего того, что было изучено в этой книге, а также то, что мы не успели рассмотреть. Грех было бы не познакомиться с конструкцией такого приложения.

В основе всего примера, созданного при помощи мастера, лежит класс `CD3DApplication`, имеющий определенный базовый набор функций и переменных. С их помощью очень легко создать "костяк" программы, ориенти-

рованной на разработку компьютерных игр. Объектно-ориентированную структуру такого приложения я и предлагаю рассмотреть. Тем более что сама структура примера и использованные в приложении классы довольно часто встречаются в разработке. Чтобы не потеряться в обилии кода, мы и рассмотрим один из таких примеров. Сразу оговорюсь, рассмотреть все в этом приложении не удастся. Для этого нужна, наверное, отдельная книга, но ключевые моменты постараемся разобрать. Если вас вдруг не устроит структура и подход к реализации примера, то что останавливает вас улучшить его?

Создание приложения при помощи мастера

Когда на *уроке 2* создавался демонстрационный пример, то были выбраны настройки по умолчанию. Сейчас я предлагаю более детально остановиться на этом моменте.

Откройте среду программирования Visual C++ .NET. Выберите в меню **File** пункт **New | Project**. Перед вами появится диалоговое окно **New Project**. В поле **Templates** выберите папку **DirectX 9 Visual C++ Wizard**, а в полях **Name** и **Location** задайте соответственно имя проекта и место его дислокации. После чего нажмите кнопку **OK**. Перед вами появится новое диалоговое окно **DirectX 9 C++ AppWizard - Urok19**, имеющее четыре активные вкладки: **Overview**, в которую вы сразу попадете, и вкладки **Project Settings**, **Direct3D Options**, **DirectInput Options**. С их помощью имеется возможность влиять на структуру создаваемого приложения. Перейдем на вкладку **Project Settings**, показанную на рис. 19.2.

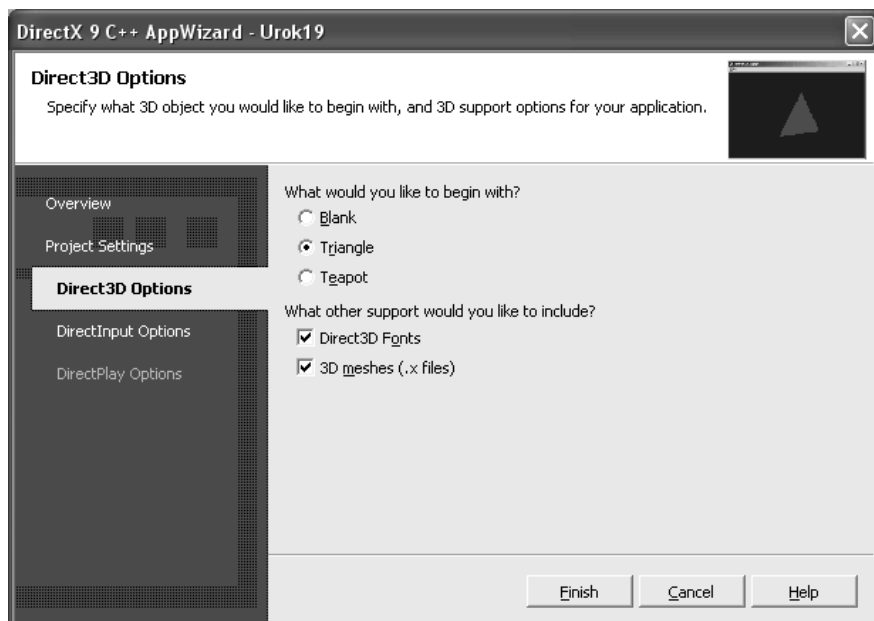
Здесь вы можете выбрать в области **What type of application would you like to create?** тип создаваемого приложения, установив один из двух флажков: **Single document window** или **MFC dialog based**. Первый флажок позволяет создать простое оконное приложение, а второй — диалоговое окно. Установите флажок в позицию **Single document window** для создания оконного приложения.

В области **What support would like to include?** задаются компоненты, которые вы хотели бы видеть в каркасе создаваемого приложения. Установите флажки напротив пунктов **Direct3D**, **DirectInput** и **DirectMusic**.

В самой нижней области **What extras would you like to include?** снимите оба флажка с пунктов **Menu bar** и **Registry access**. Эти опции нам не понадобятся.

Перейдем на вкладку **Direct3D Options**, показанную на рис. 19.3.

На этой вкладке имеются две области. Первая — **What would you like to begin with?** — позволяет создать две простые фигуры, чайник и треугольник, либо чистое приложение без моделей. Создадим простой треугольник, чайник мы уже видели.

Рис. 19.2. Вкладка **Project Settings**Рис. 19.3. Вкладка **Direct3D Options**

Во второй области — **What other support would you like to include?** — имеется возможность подключить шрифт и поддержку файлов X-формата. Воспользуемся этим предложением и поставим флажки напротив пунктов **Direct3D Fonts** и **3D meshes (.x files)**.

Перейдем на вкладку **DirectInput Options**, показанную на рис. 19.4.

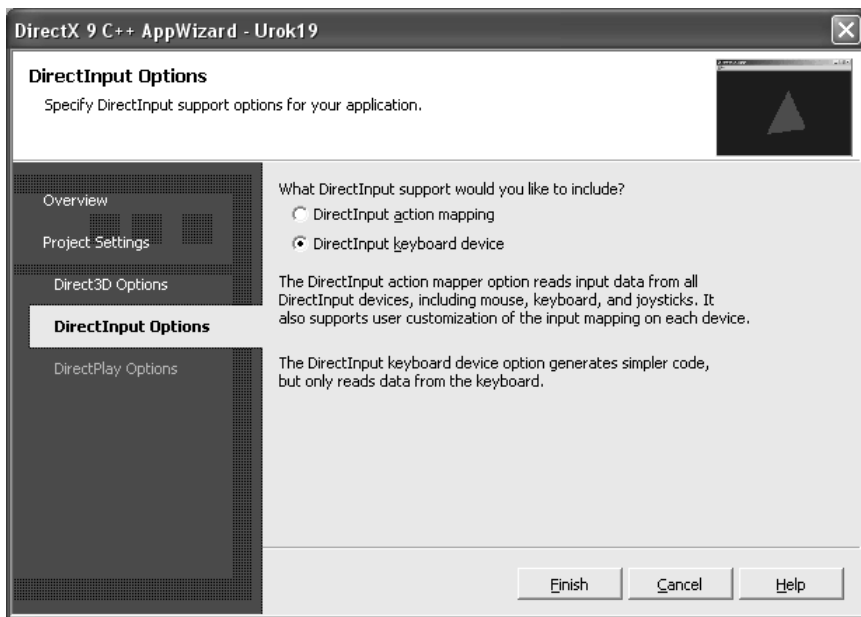


Рис. 19.4. Вкладка **DirectInput Options**

На вкладке имеется вопрос с двумя вариантами ответов. При выборе первого варианта ответа предоставляется возможность использовать созданный шаблон активных клавиш и кнопок, будь то клавиатура, мышь или джойстик, где за каждой клавишей и кнопкой уже зарезервировано определенное действие. Применим обыкновенное устройство клавиатуры, рассмотренное на *уроке 15*, и выберем переключатель **DirectInput keyboard device**. Затем нажмите на кнопку **Finish**. Это было последнее окно диалога, и в результате вы получили полноценное приложение с поддержкой всех изученных нами компонентов. В среде программирования Visual C++ .NET в окне **Solution Explorer** появится созданный проект Urok19, состоящий из двух десятков файлов. Все эти файлы лежат в общей "куче", но могут быть помещены каждый в свою папку. Файлы кода, как правило с расширением **cpp**, помещаются в папку **Source Files**, а заголовочные файлы с расширением **h** — в папку **Header Files**.

Создайте две такие папки и отсортируйте все файлы для удобства. Еще имеются файлы ресурсов, их тоже можно поместить в соответствующую папку **Header Resource**.

После компиляции примера на экране появится оконное приложение с окном синего цвета и красным треугольником. С помощью клавиш <Left>, <Right>, <Page Up> и <Page Down> треугольник можно разворачивать вокруг своей оси. При нажатии на клавишу <F2> появляется диалоговое окно, позволяющее выбрать оптимальный режим отображения информации на дисплее.

Сгенерированные классы

Теперь перейдем к структуре всего приложения. С помощью мастера был сформирован ряд классов, осуществляющих настройку и инициализацию средств DirectX 9. В табл. 19.1 приведены основные классы, их назначение и местонахождение.

Таблица 19.1. Классы

Класс	Назначение	Местонахождение
CD3DApplication	Главный класс	D3DApp.h, D3DApp.cpp
CMyD3DApplication	Наш класс	Urok19.h, Urok19.cpp
CD3DFont	Шрифт, текстуры	D3DFont.h, D3DFont.cpp
CD3DMesh	X-файлы	D3DFile.h, D3DFile.cpp
CMusicManager	Инициализация аудиосистемы	DMUtil.h, DMUtil.cpp
CMusicSegment	Воспроизведение	DMUtil.h, DMUtil.cpp
CD3DArcBall	Перемещение с помощью мыши	DMUtil.h, DMUtil.cpp
CD3DCamera	Камера	DMUtil.h, DMUtil.cpp

В каждом заголовочном файле может быть объявлено более одного класса и соответственно, в файле кода дана реализация этих классов. Поэтому предлагаю рассматривать файлы приложения на предмет наличия классов.

Файлы D3DApp.h и D3DApp.cpp

Оба файла, как вы знаете, содержат информацию о классе, в заголовочном файле происходит объявление класса, а в файле кода — его реализация, поэтому будем рассматривать оба файла как одно целое — голова без туловища жить не может.

В файлах D3DApp.h и D3DApp.cpp содержится самый главный класс CD3DApplication, создающий основные указатели на интерфейсы IDirect3D9 и IDirect3DDevice9. Там же создаются переменные, содержащие параметры представления для устройства при помощи структуры D3DPRESENT_PARAMETERS, и множество функций, осуществляющих инициализацию и настройку приложения. Чтобы все это было доступно, нужно создать свой класс, наследуемый от CD3DApplication, а поскольку мы воспользовались мастером создания приложения, то такой класс уже есть. Давайте перейдем непосредственно к его рассмотрению.

Файлы Urok19.h и Urok19.cpp

Эти два файла являются рабочими лошадками всего приложения и отражают именно ту структуру, которую вы выбрали на этапе создания примера с помощью мастера. В этих файлах находится класс CMyD3DApplication, наследуемый от CD3DApplication. В листинге 19.1 приведен оригинальный код, давайте остановимся подробно на его ключевых моментах.

Листинг 19.1. Файл Urok19.h

```
//-----
//  файл Urok19.h
//-----

#pragma once

//-----
//  подключения и константы
//-----

#define DXAPP_KEY TEXT("Urok19")

struct CUSTOMVERTEX
{
    D3DXVECTOR3 position;    // позиция вершины
    D3DXVECTOR3 normal;     // позиция нормали
};

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ | D3DFVF_NORMAL)

struct UserInput
{
    BYTE diks[256];
    BOOL bRotateUp;
    BOOL bRotateDown;
    BOOL bRotateLeft;
```

```

    BOOL bRotateRight;
    BOOL bPlaySoundButtonDown;
};
//-----
//  класс CMyD3DApplication
//-----
class CMyD3DApplication : public CD3DApplication
{
    BOOL                m_bLoadingApp;
    LPDIRECT3DVERTEXBUFFER9 m_pVB;
    CD3DFont*           m_pFont;
    LPDIRECTINPUT8       m_pDI;
    LPDIRECTINPUTDEVICE8 m_pKeyboard;
    UserInput            m_UserInput;

    FLOAT                m_fSoundPlayRepeatCountdown;
    CMusicManager*       m_pMusicManager;
    CMusicSegment*       m_pBounceSound;
    FLOAT                m_fWorldRotX;
    FLOAT                m_fWorldRotY;

protected:
    virtual HRESULT OneTimeSceneInit();
    virtual HRESULT InitDeviceObjects();
    virtual HRESULT RestoreDeviceObjects();
    virtual HRESULT InvalidateDeviceObjects();
    virtual HRESULT DeleteDeviceObjects();
    virtual HRESULT Render();
    virtual HRESULT FrameMove();
    virtual HRESULT FinalCleanup();
    virtual HRESULT ConfirmDevice(D3DCAPS9*, DWORD, D3DFORMAT);

    HRESULT RenderText();

    HRESULT InitInput(HWND hWnd);
    void UpdateInput(UserInput* pUserInput);
    void CleanupDirectInput();
    HRESULT InitAudio(HWND hWnd);

```

```
public:  
    LRESULT MsgProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);  
    CMyD3DApplication();  
    virtual ~CMyD3DApplication();  
};
```

В начале файла происходит объявление констант и разные подключения. Структура `CUSTOMVERTEX` представлена двумя переменными для позиций вершин и нормалей. Директива препроцессора `#define` устанавливает формат преобразованной вершины.

Далее идет структура `UserInput`, отвечающая за обработку нажатий клавиш при помощи массива `diks[]` и булевых переменных, проверяющих, нажата клавиша или нет.

В конце всего файла `Urok19.h` расположена полная спецификация класса `CMyD3DApplication`. Посмотрите на члены класса — это все то, что мы проходили на предыдущих уроках. Члены `m_pFont`, `m_pMusicManager` и `m_pBounceSound` объявлены как объекты классов соответственно `CD3DFont`, `CMusicManager` и `CMusicSegment`. Две переменные — `m_fWopldRotX` и `m_fWorldRotY` — отвечают за вращение по осям X и Y .

Далее идут функции класса, с помощью которых происходят инициализация, настройка, анимация, рендеринг, чистка и обработка сообщений. Если вы перейдете в файл `Urok19.cpp`, находящийся на компакт-диске в папке `Code\Urok19`, то сможете понаблюдать за процессом, происходящим внутри нашей "рабочей лошадки". Файл очень большой и печатать его листинг не имеет смысла, просто обратитесь к самому приложению. Главная входная функция `WinMain()` имеет интересный вид, по крайней мере она отличается от того, что мы делали. На самом деле все просто: создаем объект нашего класса `d3dApp` и при помощи функции `Create()`, наследуемой из класса `CD3DApplication`, поручаем ей поработать в ночную смену, проинициализировав все, что только возможно, и даже позаботиться о цикле обработки сообщений. Не об этом ли вы мечтали?

Следующим на очереди в файле `Urok19.cpp` идет конструктор, с помощью которого все указатели на интерфейсы обнуляются, а все остальные переменные задаются необходимыми значениями.

Функция `OneTimeSceneInit()` вызывается один раз в начале работы и инициализирует приложение. В данный момент происходит инициализация `DirectInput` и аудиосистемы.

Следующие две функции — `InitInput()` и `InitAudio()` — создают объект `DirectInput` и аудиосистему, и как раз вызываются в рассмотренной выше функции `OneTimeSceneInit()`.

Функция `InitDeviceObjects()` создает объект и записывает все данные в буфер вершин, предварительно заблокировав его. После окончания копирования данных буфер разблокируется. Запись, которая применяется для представления координат вершин и нормалей, по смыслу одинакова с используемой нами на уроках, и представлена в виде структуры `D3DXVECTOR(x, y, z)`.

Внутри функции `RestoreDeviceObjects()` осуществляются установки материала, освещения, матричных преобразований и текстурных уровней для объекта.

Функция `FrameMove()` производит плавную анимацию, а именно — вращение вокруг своей оси.

С помощью функции `UpdateInput()` происходит захват клавиатуры и обработка нажатий клавиш `<Left>`, `<Right>`, `<Page Up>` и `<Page Down>`.

Две функции — `Render()` и `RenderText()` — в особых комментариях не нуждаются.

Оставшиеся функции в основном следят за чисткой и производят "уборку" по окончании работы приложения.

После того как мы познакомились с классом `CMYD3DApplication`, я думаю, общая структура приложения вам должна быть понятна. Для того чтобы не производить черновую работу, вы создаете свой класс, наследуемый от `CD3DApplication`, и пользуетесь всем его содержимым. Параллельно этим действиям создается еще несколько классов, отвечающих за шрифт и текстуры, устройства ввода, X-файлы и музыку. А создавая объекты классов, вы используете все возможности в реализации данного класса.

Файл `D3DFont.h` и `D3DFont.cpp`

В этих файлах содержится один обобщенный класс `CD3DFont` для шрифта и текстурирования. Сам класс выглядит следующим образом:

```
class CD3DFont
{
    TCHAR m_strFontName[80];
    DWORD m_dwFontHeight;
    DWORD m_dwFontFlags;

    LPDIRECT3DDEVICE9 m_pd3DDevice;
    LPDIRECT3DTEXTURE9 m_pTexture;
    LPDIRECT3DVERTEXBUFFER9 m_pVB;
    DWORD m_dwTexWidth;
    DWORD m_dwTexHeight;
    FLOAT m_fTextScale;
```

```

FLOAT          m_fTexCoords[128-32][4];
DWORD          m_dwSpacing;

LPDIRECT3DSTATEBLOCK9 m_pStateBlockSaved;
LPDIRECT3DSTATEBLOCK9 m_pStateBlockDrawText;

public:

HRESULT DrawText(FLOAT x, FLOAT y, DWORD dwColor,
                 const TCHAR* strText, DWORD dwFlags = 0L);
HRESULT DrawTextScaled(FLOAT x, FLOAT y, FLOAT z,
                       FLOAT fXScale, FLOAT fYScale, DWORD dwColor,
                       const TCHAR* strText, DWORD dwFlags = 0L);
HRESULT Render3DText(const TCHAR* strText, DWORD dwFlags = 0L);

HRESULT GetTextExtent(const TCHAR* strText, SIZE* pSize);

HRESULT InitDeviceObjects(LPDIRECT3DDEVICE9 pd3DDevice);
HRESULT RestoreDeviceObjects();
HRESULT InvalidateDeviceObjects();
HRESULT DeleteDeviceObjects();

CD3DFont(const TCHAR* strFontName, DWORD dwHeight, DWORD dwFlags=0L);
~CD3DFont();
};

```

В спецификации класса происходит объявление указателей на интерфейсы для работы с устройством Direct3D, буфером вершин и текстурой. Все остальные переменные представляют свойства шрифта. Функции DrawText(), DrawTextScaled() и Render3DText() отвечают за вывод текста на экран. Каждая из функций делает это по-разному, со своим так называемым набором "фишек".

Файлы D3DFile.h и D3DFile.cpp

В этих файлах содержатся сразу три класса: CD3DMesh, CD3DFrame и класс CD3DFile, наследуемый от CD3DFrame. В листинге 19.2 показан текст файла D3DFile.h.

Листинг 19.2. Файл D3DFile.h

```

#ifndef D3DFILE_H
#define D3DFILE_H
#include <tchar.h>
#include <d3d9.h>
#include <d3dx9.h>
//*****
class CD3DMesh
{
public:
    TCHAR                m_strName[512];

    LPD3DXMESH            m_pSysMemMesh;
    LPD3DXMESH            m_pLocalMesh;

    DWORD                m_dwNumMaterials;
    D3DMATERIAL9*        m_pMaterials;
    LPDIRECT3DTEXTURE9*  m_pTextures;
    bool                 m_bUseMaterials;

public:
    // рендеринг
    HRESULT Render(LPDIRECT3DDEVICE9 pd3DDevice,
                  bool bDrawOpaqueSubsets = true,
                  bool bDrawAlphaSubsets = true);

    // меш
    LPD3DXMESH GetSysMemMesh(){return m_pSysMemMesh;}
    LPD3DXMESH GetLocalMesh(){return m_pLocalMesh;}

    // опции рендеринга
    void UseMeshMaterials(bool bFlag){m_bUseMaterials = bFlag;}
    HRESULT SetFVF(LPDIRECT3DDEVICE9 pd3DDevice, DWORD dwFVF);

    // инициализация
    HRESULT RestoreDeviceObjects(LPDIRECT3DDEVICE9 pd3DDevice);
    HRESULT InvalidateDeviceObjects();

```

```

// создание и удаление
HRESULT Create(LPDIRECT3DDEVICE9 pd3DDevice, TCHAR* strFilename);
HRESULT Create(LPDIRECT3DDEVICE9 pd3DDevice,
               LPDIRECTXFILEDATA pFileData);
HRESULT Destroy();

CD3DMesh(TCHAR* strName = _T("CD3DFile_Mesh"));
virtual ~CD3DMesh();
};

//*****
class CD3DFrame
{
public:
    TCHAR        m_strName[512];
    D3DXMATRIX m_mat;
    CD3DMesh* m_pMesh;

    CD3DFrame* m_pNext;
    CD3DFrame* m_pChild;

public:
    // матрицы
    void SetMatrix(D3DXMATRIX* pmat){m_mat = *pmat;}
    D3DXMATRIX* GetMatrix() {return &m_mat;}

    CD3DMesh* FindMesh(TCHAR* strMeshName);
    CD3DFrame* FindFrame(TCHAR* strFrameName);
    bool EnumMeshes(bool (*EnumMeshCB) (CD3DMesh*,void*), void* pContext);

    HRESULT Destroy();
    HRESULT RestoreDeviceObjects(LPDIRECT3DDEVICE9pd3DDevice);
    HRESULT InvalidateDeviceObjects();
    HRESULT Render(LPDIRECT3DDEVICE9 pd3DDevice,
                  bool bDrawOpaqueSubsets = true,
                  bool bDrawAlphaSubsets = true,
                  D3DXMATRIX* pmatWorldMartix = NULL);

    CD3DFrame(TCHAR* strName = _T("CD3DFile_Frame"));
    virtual ~CD3DFrame();
};

```

```
//*****

class CD3DFile : public CD3DFrame
{
    HRESULT LoadMesh(LPDIRECT3DDEVICE9 pd3DDevice,
                     LPDIRECTXFILEDATA pFileData,
                     CD3DFrame* pParentFrame);
    HRESULT LoadFrame(LPDIRECT3DDEVICE9 pd3DDevice,
                     LPDIRECTXFILEDATA pFileData,
                     CD3DFrame* pParentFrame);

public:
    HRESULT Create(LPDIRECT3DDEVICE9 pd3DDevice, TCHAR* strFilename);
    HRESULT CreateFromResource(LPDIRECT3DDEVICE9 pd3DDevice,
                              TCHAR* strResource, TCHAR* strType);

    HRESULT Render(LPDIRECT3DDEVICE9 pd3DDevice,
                  D3DXMATRIX* pmatWorldMatrix = NULL);

    CD3DFile() : CD3DFrame(_T("CD3DFile_Root")) {}
};

#endif
```

Все три класса реализуют работу с файлами X-формата. Если вы изучали *урок 12*, то вам не составит никакого труда разобраться с этими классами. Вся реализация трех классов основана на нахождении X-файла в каталоге, загрузке его в приложение и правильном представлении на экране. Эти действия планомерно разделены на три класса, создав объекты которых, вы сможете манипулировать X-файлами.

Файлы DMUtil.h и DMUtil.cpp

Эти файлы отвечают за звуковое оформление приложения и содержат в себе объявления сразу четырех классов: `CMusicManager`, `CMusicSegment`, `CMusicScript` и `C3DmusicSegment`, наследуемого от класса `CMusicSegment`.

Класс `CMusicManager` инициализирует всю аудиосистему и подгружает звуковой файл в приложение. Класс `CMusicSegment` в большей степени ориентирован на воспроизведение звукового файла, остановку проигрывания, заикливание воспроизведения файла. Два оставшихся класса отвечают за 3D-звук.

Пожалуй, это все ключевые классы этого приложения. Единственное, хотелось бы упомянуть еще о двух классах — это `CD3DArcBall` и `CD3Dcamera`, находящиеся в файлах `D3DUtil.h` и `D3DUtil.cpp`. Эти файлы содержат так называемые вспомогательные функции и классы для программирования графики средствами `Direct3D`.

Класс `CD3DArcBall` решает за нас задачу по перемещению объекта и камеры на основе движения мыши. Помните, как в играх типа "стрелялки"? Вы перемещаете мышь, и соответственно меняется угол обзора. Это можно реализовать при помощи класса `CD3DArcBall`. Класс `CD3DCamera`, как видно из названия, реализует работу с камерой на основе матрицы вида, которая, как вы помните, отвечает за местонахождение точки просмотра. Посмотрите на спецификацию этого класса:

```
class CD3DCamera
{
    D3DXVECTOR3  m_vEyePt;
    D3DXVECTOR3  m_vLookatPt;
    D3DXVECTOR3  m_vUpVec;

    D3DXVECTOR3  m_vView;
    D3DXVECTOR3  m_vCross;

    D3DXMATRIX   m_matView;
    D3DXMATRIX   m_matBillboard;

    FLOAT        m_fFOV;
    FLOAT        m_fAspect;
    FLOAT        m_fNearPlane;
    FLOAT        m_fFarPlane;
    D3DXMATRIX   m_matProj;

public:

    D3DXVECTOR3  GetEyePt() {return m_vEyePt;}
    D3DXVECTOR3  GetLookatPt() {return m_vLookatPt;}
    D3DXVECTOR3  GetUpVec() {return m_vUpVec;}
    D3DXVECTOR3  GetViewDir() {return m_vView;}
    D3DXVECTOR3  GetCross() {return m_vCross;}

    FLOAT  GetFOV() {return m_fFOV;}
    FLOAT  GetAspect() {return m_fAspect;}
```

```

FLOAT GetNearPlane() {return m_fNearPlane;}
FLOAT GetFarPlane() {return m_fFarPlane;}

D3DXMATRIX GetViewMatrix() {return m_matView;}
D3DXMATRIX GetBillboardMatrix() {return m_matBillboard;}
D3DXMATRIX GetProjMatrix() {return m_matProj;}

VOID SetViewParams (D3DXVECTOR3&vEyePt,
                   D3DXVECTOR3&vLookatPt,
                   D3DXVECTOR3& vUpVec);
VOID SetProjParams (FLOAT fFOV, FLOAT fAspect,
                   FLOAT fNearPlane, FLOAT fFarPlane);

CD3DCamera();
};
```

Обобщая этот урок, хочется заметить, что вся структура реализации модели классов при помощи мастера несколько громоздка. Но она имеет очень большие возможности, благодаря которым вы лишены счастья изобретать велосипед, и все, что нужно, — это только разобраться в управлении такого велосипеда, и можно смело отправляться в путешествие. С другой стороны, создание своей структуры классов выглядит тоже неплохо. В этом случае вы разрабатываете и реализуете только то, что необходимо вам для решения четко определенной концепции задачи, и ничего лишнего. При желании, чтобы не описывать какие-то элементарные вещи, всегда можно воспользоваться стандартными классами из Direct3D, создав объекты необходимых классов в своем приложении. Поэтому скорее всего некий симбиоз обоих вариантов реализации структуры приложения может выглядеть более предпочтительнее.

Итоги урока

Этот урок, как мне кажется, дал большую пищу для размышления в плане общей модели реализации классов. Но на самом деле любой путь, какой бы вы не избрали, будет правильным. Каких-то определенных правил на самом деле не существует, имеются только общие концепции и ничего больше.

Заключение

Подводя итоги всей книги, хотелось бы надеяться, что вам был интересен и, главное, понятен весь предложенный материал. Изучение такой тяжелой и большой темы, как DirectX, требует определенных познаний в математике, да и в самом языке программирования C++. В большом объеме книги был рассмотрен компонент Direct3D. В отличие от DirectInput, DirectMusic и DirectSound, описания этого интерфейса в российских книжных изданиях нет и тем ценнее данная информация. Direct3D — это очень мощная система, имеющая только около полутысячи всевозможных функций для работы с трехмерной графикой. В книге мы рассмотрели самые нужные и основные функции, классы, структуры и интерфейсы, работа без которых с Direct3D просто невозможна. С такого знакомства начинается каждый программист, желающий научиться использовать библиотеку DirectX.

Рассмотреть все в одной книге невозможно, а значит, продолжение следует...



ПРИЛОЖЕНИЯ

Приложение 1



Справочная информация

В этом приложении находится справочная информация по ряду необходимых интерфейсов, функций, типов, структур и макросов библиотеки DirectX 9.

Вся информация разбита на следующие разделы:

□ DirectX Graphics

- Интерфейсы DirectX Graphics
- Функции DirectX Graphics
- Структуры DirectX Graphics
- Типы DirectX Graphics
- Макросы DirectX Graphics

□ DirectInput

- Интерфейсы DirectInput
- Функции DirectInput
- Структуры DirectInput

□ DirectMusic

- Интерфейсы DirectMusic
- Функции DirectMusic

□ DirectSound

- Интерфейсы DirectSound
- Функции DirectSound
- Структуры DirectSound

Весь справочный материал основан на оригинальной документации по DirectX 9 от корпорации Microsoft.

DirectX Graphics

Компонент для работы с графикой, содержащий набор функций, интерфейсов, структур, типов, макросов. Некоторые из них рассмотрены в данном приложении.

Интерфейсы DirectX Graphics

В данном списке приведены интерфейсы DirectX Graphics с их кратким описанием:

- ❑ `IDirect3D9` — самый главный интерфейс, из него наследуются все остальные интерфейсы. Это первый объект, который должен создаваться;
- ❑ `IDirect3DDevice9` — интерфейс, отвечающий за устройство Direct3D;
- ❑ `IDirect3DVertexBuffer9` — необходим для работы с буфером вершин;
- ❑ `IDirect3DIndexBuffer9` — отвечает за индексный буфер;
- ❑ `IDirect3DTexture9` — работает с текстурами;
- ❑ `IDirect3DVertexShader9` — обеспечивает работу с вершинными шейдерами;
- ❑ `IDirect3DBaseTexture9` — манипулирует ресурсами текстуры, включая объемные и кубические текстуры;
- ❑ `IDirect3DCubeTexture9` — манипулирует кубическими текстурами;
- ❑ `IDirect3DPixelShader9` — интерфейс для работы с пиксельными шейдерами;
- ❑ `IDirect3DQuery9` — выполняет асинхронные запросы на драйвер;
- ❑ `IDirect3DResource9` — осуществляет запрос и подготовку устройства;
- ❑ `IDirect3DSurface9` — служит для запроса и подготовки поверхности;
- ❑ `IDirect3DSwapChain9` — производит манипуляции с цепью обмена;
- ❑ `IDirect3DVertexDeclaration9` — устанавливает декларации;
- ❑ `ID3DXAnimationController` — диспетчер анимаций;
- ❑ `ID3DXAnimationSet` — осуществляет различные установки анимаций;
- ❑ `ID3DXBaseEffect` — обеспечивает установки и параметры эффектов;
- ❑ `ID3DXBuffer` — интерфейс буфера данных;
- ❑ `ID3DXMesh` — интерфейс для работы с мешем;
- ❑ `ID3DXRenderToSurface` — обобщает процесс рендеринга на поверхности.

Функции DirectX Graphics

В этом разделе перечислены наиболее часто встречающиеся функции при программировании приложений с использованием DirectX Graphics.

Direct3DCreate9()

Создает объект Direct3D. Функция описывается таким образом:

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

Параметр:

- `SDKVersion` — всегда содержит макрос `D3D_SDK_VERSION`, возвращающий текущую версию DirectX.

IDirect3D9::CreateDevice

Создает устройство Direct3D, т. е. представляет видеоадаптер. Функция описывается таким образом:

```
HRESULT CreateDevice(  
    UINT                Adapter,  
    D3DDEVTYPE          DeviceType,  
    HWND                hFocusWindow,  
    DWORD                BehaviorFlags,  
    D3DPRESENT_PARAMETERS* pPresentationParameters,  
    IDirect3DDevice9**    ppReturnedDeviceInterface  
);
```

Параметры:

- `Adapter` — это ваша видеокарта. Обычно на компьютере стоит один видеоадаптер, поэтому используется значение `D3DADAPTER_DEFAULT`, т. е. по умолчанию первичный видеоадаптер;
- `DeviceType` — член перечисляемого типа `D3DDEVTYPE`, устанавливает тип устройства;
- `hFocusWindow` — в этом параметре необходимо установить дескриптор главного окна;
- `BehaviorFlags` — указывает, как будет происходить обработка вершин. Делается это с помощью комбинации одного или более флагов;
- `pPresentationParameters` — указатель на необходимые параметры структуры `D3DPRESENT_PARAMETERS`;
- `ppReturnedDeviceInterface` — указатель на интерфейс `IDirect3DDevice9`.

IDirect3D9::GetAdapterDisplayMode

Извлекает установленный режим дисплея. Функция описывается таким образом:

```
HRESULT GetAdapterDisplayMode(
    UINT          Adapter,
    DISPLAYMODE*  pMode);
```

Параметры:

- ❑ **Adapter** — видеоадаптер дисплея, значение `D3DADAPTER_DEFAULT` использует первичный адаптер по умолчанию;
- ❑ **pMode** — это указатель на структуру `DISPLAYMODE`, с помощью которой описывается текущий режим адаптера.

IDirect3DDevice9::Clear

Очищает различные буферы цветом. Функция описывается таким образом:

```
HRESULT Clear(
    DWORD          Count,
    CONST D3DRECT* pRects,
    DWORD          Flags,
    D3DCOLOR       Color,
    float          Z,
    DWORD          Stencil
);
```

Параметры:

- ❑ **Count** — счетчик, указывает число прямоугольников, подвергающихся очистке. Вся область рендеринга состоит из массива прямоугольников. Если указать значение 0, то будет использована вся поверхность, т. е. весь массив прямоугольников или вся область рендеринга;
- ❑ **pRects** — указатель на структуру `D3DRECT`, описывающий массив очищаемых прямоугольников;
- ❑ **Flags** — параметр, определяющий флаги, которые указывают, какие из поверхностей должны быть очищены;
- ❑ **Color** — в этом параметре используется макрос цвета `D3DCOLOR_XRGB()`;
- ❑ **Z** — этот параметр задает значения для Z-буфера;
- ❑ **Stencil** — значение для буфера трафарета может быть в диапазоне от 0 до 2^{n-1} , где n — разрядная глубина буфера трафарета.

IDirect3DDevice9::CreateVertexBuffer

Создает буфер вершин. Функция описывается таким образом:

```
HRESULT CreateVertexBuffer(  
    UINT                Length,  
    DWORD               Usage,  
    DWORD               FVF,  
    D3DPOOL             Pool,  
    IDirect3DVertexBuffer9** ppVertexBuffer,  
    HANDLE*             pHandle);
```

Параметры:

- ❑ `Length` — это размер буфера вершин в байтах;
- ❑ `Usage` — способ применения создаваемого буфера вершин. В большинстве случаев ставится 0, но можно воспользоваться одной или более константами `D3DUSAGE`. Правда, это значение придется согласовывать с параметром `BehaviorFlags` из `IDirect3D9::CreateDevice`;
- ❑ `FVF` — комбинация флагов `D3DFVF`, необходимых для установки формата вершин;
- ❑ `Pool` — член перечисляемого типа `D3DPOOL`, описывает класс памяти;
- ❑ `ppVertexBuffer` — это указатель, в котором будет храниться адрес создаваемого буфера вершин;
- ❑ `pHandle` — всегда используйте нулевое значение, это зарезервированный параметр.

IDirect3DDevice9::CreateVertexDeclaration

Создает декларацию вершинного шейдера. Функция описывается таким образом:

```
HRESULT CreateVertexDeclaration(  
    CONST D3DVERTEXELEMENT9* pVertexElements,  
    IDirect3DVertexDeclaration9** ppDecl  
);
```

Параметры:

- ❑ `pVertexElements` — переменная, содержащая декларации шейдера;
- ❑ `ppDecl` — указатель на интерфейс `IDirect3DVertexDeclaration9`.

IDirect3DDevice9::CreateVertexShader

Создает вершинный шейдер. Функция описывается таким образом:

```
HRESULT CreateVertexShader(
    const DWORD*          pFunction,
    IDirect3DVertexShader9** ppShader
);
```

Параметры:

- ❑ `pFunction` — указатель, содержащий данные, возвращаемые функцией `ID3DXBuffer::GetBufferPointer`;
- ❑ `ppShader` — указатель на интерфейс `IDirect3DVertexShader9`.

IDirect3DDevice9::DrawPrimitive

Рисует последовательность неиндексированных примитивов. Функция описывается таким образом:

```
HRESULT DrawPrimitive(
    D3DPRIMITIVETYPE PrimitiveType,
    UINT              StartVertex,
    UINT              PrimitiveCount)
```

Параметры:

- ❑ `PrimitiveType` — это перечисляемый тип члена `D3DPRIMITIVETYPE`, описывает тип примитива, выводимого на экран;
- ❑ `StartVertex` — индекс первой вершины для загрузки;
- ❑ `PrimitiveCount` — своего рода счетчик, указывающий на количество выводимых треугольников на экран. Рисуем один треугольник, значит, ставим цифру 1, два треугольника — значит, цифра 2 и т. д.

IDirect3DDevice9::GetDirect3D

Возвращает интерфейс на созданное устройство. Функция описывается таким образом:

```
HRESULT GetDirect3D(IDirect3D9** ppD3D9);
```

Параметр:

- ❑ `ppD3D9` — адрес указателя на интерфейс `IDirect3D9`, который является интерфейсом объекта `Direct3D`.

IDirect3DDevice9::LightEnable

Включает или отключает осветительные параметры устройства. Функция описывается таким образом:

```
HRESULT LightEnable(  
    DWORD LightIndex,  
    BOOL bEnable);
```

Параметры:

- ❑ **LightIndex** — в этом параметре указывается индекс, определяющий свойства света. Отсчет идет от 0;
- ❑ **bEnable** — булева константа, имеющая два значения — FALSE и TRUE, соответственно запрещающая и разрешающая использовать освещение в значении **LightIndex**.

IDirect3DDevice9::Present

Представляет содержимое буфера на дисплей. Функция описывается таким образом:

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

Параметры:

- ❑ **pSourceRect** — указатель на структуру RECT исходной поверхности. Если использовать значение NULL, то будет представлена полная исходная поверхность;
- ❑ **pDestRect** — указатель на структуру RECT поверхности адресата;
- ❑ **hDestWindowOverride** — указатель на окно адресата, чья клиентская область взята как адрес для этого параметра представления. Установив значение NULL, мы тем самым говорим, что используем поле **hWndDeviceWindow** в структуре **D3DPRESENT_PARAMETERS**;
- ❑ **pDirtyRegion** — последний параметр, "отголосок" прежних версий DirectX и уже не используется. Оставлен просто для совместимости, поэтому устанавливается всегда в значение NULL.

IDirect3DDevice9::SetLight

Назначает набор осветительных свойств. Функция описывается таким образом:

```
HRESULT SetLight(  
    DWORD          Index,  
    CONST D3LIGHT9* pLight);
```

Параметры:

- ❑ Index — это индекс, устанавливающий свойства света. Отсчет начинается с 0;
- ❑ pLight — это и есть созданный источник света, указатель на структуру D3LIGHT9.

IDirect3DDevice9::SetMaterial

Устанавливает материальные свойства. Функция описывается таким образом:

```
HRESULT SetMaterial(CONST D3DMATERIAL9* pMaterial);
```

Параметр:

- ❑ pMaterial — указатель на структуру D3DMATERIAL9, результат всей операции по установке свойств поверхности объекта.

IDirect3DDevice9::SetRenderState

Устанавливает единственное устройство рендеринга. Функция описывается таким образом:

```
HRESULT SetRenderState(  
    D3DRENDERSTATETYPE State,  
    DWORD                Value);
```

Параметры:

- ❑ State — значением для этой переменной может быть любой из членов перечисляемого типа D3DRENDERSTATETYPE, устанавливающий вид работы для устройства pDirect3DDevice;
- ❑ Value — это значение полностью зависит от параметра State. Определяет как параметр State будет работать.

IDirect3DDevice9::SetStreamSource

Связывает буфер вершин с потоком данных. Функция описывается таким образом:

```
HRESULT SetStreamSource(  
    UINT                StreamNumber,
```

```
IDirect3DVertexBuffer* pStreamData,
UINT                    OffsetInBytes,
UINT                    Stride)
```

Параметры:

- ❑ `StreamNumber` — определяет поток данных в диапазоне от 0 до -1;
- ❑ `pStreamData` — это указатель на интерфейс `IDirect3DVertexBuffer`, созданный для представления буфера вершин, который связывается с определенным потоком данных;
- ❑ `OffsetInBytes` — смещение от начала потока до начала данных вершин. Измеряется в байтах. Здесь вы указываете, с какой вершины начинается вывод. Если поставить 0, то вывод будет происходить от первой вершины;
- ❑ `Stride` — это своего рода "шаг в байтах", от одной вершины до другой.

IDirect3DDevice9::SetTexture

Назначает текстуру для устройства. Функция описывается таким образом:

```
HRESULT SetTexture(
    DWORD                    Stage,
    IDirect3DBaseTexture9* pTexture);
```

Параметры:

- ❑ `Stage` — это идентификатор, определенный для каждой текстуры. Один объект может иметь до восьми текстур, поэтому каждая текстура должна иметь свой идентификатор. Отсчет начинается с 0, заканчивается 7;
- ❑ `pTexture` — указатель на интерфейс `IDirect3DBaseTexture9`, где и находится сама загружаемая текстура.

IDirect3DDevice9::SetTextureStageState

Устанавливает состояние для текущей текстуры. Функция описывается таким образом:

```
HRESULT SetTextureStageState(
    DWORD                    Stage,
    D3DTEXTURESTAGESTATETYPE Type,
    DWORD                    Value);
```

Параметры:

- ❑ `Stage` — идентификатор, определенный для текстуры в границах от 0 до 7 (еще говорят, текстурный уровень);
- ❑ `Type` — член перечисляемого типа `D3DTEXTURESTAGESTATETYPE`, служит для установки состояния загружаемой текстуры;

- `Value` — эта переменная определяет текстурную операцию для состояния текстуры, установленной в параметре `Type`.

IDirect3DDevice9::SetTransform

Устанавливает единственное устройство преобразования. Функция описывается таким образом:

```
HRESULT SetTransform(
    D3DTRANSFORMSTATETYPE State,
    CONST D3DMATRIX*      pMatrix);
```

Параметры:

- `State` — этот параметр указывает на происходящее преобразование;
- `pMatrix` — указатель на матрицу, установленную как текущее преобразование.

IDirect3DDevice9::SetVertexDeclaration

Устанавливает декларацию вершинного шейдера. Функция описывается таким образом:

```
HRESULT SetVertexDeclaration(IDirect3DVertexDeclaration9* pDecl);
```

Параметр:

- `pDecl` — указатель на интерфейс `IDirect3DVertexDeclaration9`.

IDirect3DDevice9::SetVertexShaderConstantF

Устанавливает константы с плавающей точкой для вершины шейдера. Функция описывается таким образом:

```
HRESULT SetVertexShaderConstantF(
    UINT          StartRegister,
    CONST float*  pConstantData,
    UINT          Vector4fCount
);
```

Параметры:

- `StartRegister` — первый номер регистра, в который будут записаны данные;
- `pConstantData` — указатель, содержащий данные для записи в регистр;
- `Vector4fCount` — количество необходимых регистров для размещения данных.

IDirect3DBaseTexture9::SetPrivateData

Связывание данных. Функция описывается таким образом:

```
HRESULT SetPrivateData(  
    REFGUID    refguid,  
    const void* pData,  
    DWORD      SizeOfData,  
    DWORD      Flags);
```

Параметры:

- ❑ `refguid` — глобально-уникальный идентификатор;
- ❑ `pData` — указатель на буфер, содержащий данные, которые нужно связать с ресурсом;
- ❑ `SizeOfData` — размер буфера в байтах;
- ❑ `Flags` — флаги, с помощью которых описывается тип данных.

IDirect3DVertexBuffer::Lock

Блокирует указанный диапазон данных в буфере вершин. Функция описывается таким образом:

```
HRESULT Lock(  
    UINT    OffsetToLock,  
    UINT    SizeToLock,  
    VOID**  ppbData,  
    DWORD   Flags)
```

Параметры:

- ❑ `OffsetToLock` — указывает на границу данных вершин для блокировки. Измеряется в байтах. Для того чтобы заблокировать весь буфер, ставится значение 0. Это своего рода смещение, показывающее, от какого значения нужно заблокировать буфер;
- ❑ `SizeToLock` — этим параметром определяется размер блокировки буфера. В первом параметре мы говорим, откуда блокировать, а во втором — на сколько блокировать. Кстати, оба параметра можно выставить в 0, этим вы заблокируете весь буфер вершин;
- ❑ `ppbData` — это адрес указателя на указатель с данными буфера вершин, необходимый для использования функции копирования `memcpy()`. Эту временную переменную необходимо создать перед тем, как задействовать функцию `Lock()`;
- ❑ `Flags` — описывает тип осуществляемой блокировки. В большинстве случаев это комбинация из различных флагов (до 5) для описания типа блокировки.

ID3DXBaseEffect::SetVertexShader

Устанавливает массив векторов. Функция описывается таким образом:

```
HRESULT SetVertexShader(IDirect3DVertexShader9* pShader);
```

Параметр:

□ `pShader` — указатель на интерфейс `IDirect3DVertexShader9`.

ID3DXBaseMesh::DrawSubset

Рисует мэш. Функция описывается таким образом:

```
HRESULT DrawSubset(DWORD AttribId);
```

Параметр:

□ `AttribId` — это атрибут, рисующий установленную часть мэша. Весь мэш рисуется как бы небольшими областями данных посредством задания индекса атрибута.

ID3DXFont::DrawText

Выводит форматированный текст. Функция описывается таким образом:

```
HRESULT DrawText(
    LPCSTR    pString,
    INT       Count,
    LPRECT    pRect,
    DWORD     Format,
    D3DCOLOR  Color);
```

Параметры:

- `pString` — указатель на строку текста;
- `Count` — это счетчик, определяющий количество символов в параметре `pString`. Чтобы не производить постоянный подсчет символов, достаточно поставить `-1`, и тогда подсчет будет производиться автоматически;
- `pRect` — адрес прямоугольной области дисплея, в которой выводится текст;
- `Format` — с помощью этого значения происходит форматирование текста внутри области вывода. Параметр не влияет на ширину, высоту и другие характеристики шрифта, а форматирует выводимый текст относительно сторон прямоугольника. А именно двигает текст внутри области вывода вверх, вправо, влево и т. д. Это достигается путем задания заранее определенных флагов;
- `Color` — последний параметр, определяющий цвет выводимого текста на экран, член структуры `D3DCOLOR`.

ID3DXSkinInfo::SetFVF

Устанавливает формат вершин FVF. Функция описывается таким образом:

```
HRESULT SetFVF(DWORD FVF);
```

Параметр:

□ FVF — формат вершин.

D3DXAssembleShader()

Сборка шейдера. Функция описывается таким образом:

```
HRESULT WINAPI D3DXAssembleShader(  
    LPCTSTR          pSrcData,  
    UINT             SrcDataLen,  
    CONST D3DXMACRO* pDefines,  
    LPD3DXINCLUDE     pInclude,  
    DWORD            Flags,  
    LPD3DXBUFFER*     ppShader,  
    LPD3DXBUFFER*     ppErrorMsgs  
);
```

Параметры:

- pSrcData — указатель на буфер памяти, который содержит данные шейдера;
- SrcDataLen — длина данных в байтах;
- pDefines — указатель на препроцессор;
- pInclude — дополнительный специфический указатель на интерфейс ID3DXInclude. Если в программном коде используется директива #include, это значение выставляется в ноль;
- Flags — определяет опции для компиляции D3DXSHADER;
- ppShader — возвращает буфер, содержащий созданный шейдер. Этот буфер содержит откомпилированный код шейдера;
- ppErrorMsgs — возвращает листинг ошибок и предупреждений, с которыми столкнулись в течение компиляции.

D3DXAssembleShaderFromFile()

Сборка шейдера из файла. Функция описывается таким образом:

```
HRESULT WINAPI D3DXAssembleShaderFromFile(  
    LPCTSTR          pSrcFile,  
    CONST D3DXMACRO* pDefines,
```

```

    LPD3DXINCLUDE    pInclude,
    DWORD            Flags,
    LPD3DXBUFFER*    ppShader,
    LPD3DXBUFFER*    ppErrorMsgs
);

```

Параметры:

- ❑ `pSrcFile` — указатель, содержащий имя файла;
- ❑ `pDefines` — указатель на препроцессор;
- ❑ `pInclude` — дополнительный указатель интерфейса `ID3DXInclude`. Если в программном коде используется директива `#include`, это значение выставляется в ноль;
- ❑ `Flags` — флаги, идентифицирующие шейдер;
- ❑ `ppShader` — возвращает буфер, содержащий созданный шейдер. Этот буфер содержит откомпилированный код шейдера;
- ❑ `ppErrorMsgs` — возвращает листинг ошибок и предупреждений, с которыми столкнулись в течение компиляции.

D3DXAssembleShaderFromResource()

Сборка шейдера из ресурса. Функция описывается таким образом:

```

HRESULT WINAPI D3DXAssembleShaderFromResource (
    HMODULE            hSrcModule,
    LPCTSTR            pSrcResource,
    CONST D3DXMACRO*   pDefines,
    LPD3DXINCLUDE      pInclude,
    DWORD              Flags,
    LPD3DXBUFFER*      ppShader,
    LPD3DXBUFFER*      ppErrorMsgs
);

```

Параметры:

- ❑ `hSrcModule` — значение `NULL` устанавливает использование текущего модуля;
- ❑ `pSrcResource` — указатель, определяющий имя ресурса;
- ❑ `pDefines` — указатель на препроцессор;
- ❑ `pInclude` — дополнительный указатель на интерфейс `ID3DXInclude`;
- ❑ `Flags` — опции компиляции для `D3DXSHADER`;
- ❑ `ppShader` — возвращает буфер, содержащий созданный шейдер. Этот буфер содержит откомпилированный код шейдера;

- ❑ `ppErrorMsgs` — возвращает листинг ошибок и предупреждений, с которыми столкнулись в течение компиляции.

D3DXCleanMesh()

Чистит мэш. Функция описывается таким образом:

```
HRESULT D3DXCleanMesh(  
    LPD3DXMESH    pMeshIn,  
    const DWORD*   pAdjacencyIn,  
    LPD3DXMESH*    ppMeshOut,  
    DWORD*         pAdjacencyOut,  
    LPD3DXBUFFER*  ppErrorsAndWarnings  
);
```

Параметры:

- ❑ `pMeshIn` — указатель на интерфейс `ID3DXMesh`, представляющий мэш, который нужно очистить;
- ❑ `pAdjacencyIn` — указатель на массив значений трех лицевых граней в меше для чистки;
- ❑ `ppMeshOut` — адрес указателя на интерфейс `ID3DXMesh`, представляющий возвращенный чистый мэш;
- ❑ `pAdjacencyOut` — указатель на массив значений трех лицевых граней;
- ❑ `ppErrorsAndWarnings` — возвращает ошибку либо предостережение.

D3DXColorModulate()

Смешивает два цвета. Функция описывается таким образом:

```
D3DXCOLOR* D3DXColorModulate(  
    D3DXCOLOR*    pOut,  
    CONST D3DXCOLOR* pC1,  
    CONST D3DXCOLOR* pC2  
);
```

Парамеры:

- ❑ `pOut` — указатель на структуру `D3DXCOLOR`, является результатом операции по смешиванию двух цветов, находящихся в параметрах `pC1` и `pC2`;
- ❑ `pC1` — указатель на исходную структуру `D3DXCOLOR`;
- ❑ `pC2` — указатель на исходную структуру `D3DXCOLOR`.

D3DXCompileShader()

Компилирует файл шейдера. Функция описывается таким образом:

```
HRESULT WINAPI D3DXCompileShader(
    LPCTSTR                pSrcData,
    UINT                   srcDataLen,
    CONST D3DXMACRO*       pDefines,
    LPD3DXINCLUDE          pInclude,
    LPCTSTR                pFunctionName,
    LPCTSTR                pTarget,
    DWORD                  Flags,
    LPD3DXBUFFER*          ppShader,
    LPD3DXBUFFER*          ppErrorMsgs,
    LPD3DXSHADER_CONSTANTTABLE* ppConstantTable
);
```

Параметры:

- ❑ `pSrcData` — указатель, содержащий шейдер;
- ❑ `srcDataLen` — длина данных в байтах;
- ❑ `pDefines` — указатель на препроцессор;
- ❑ `pInclude` — дополнительный указатель интерфейса `ID3DXInclude`;
- ❑ `pFunctionName` — указатель, содержащий имя входной функции;
- ❑ `pTarget` — указатель на строку, которая содержит цель компиляции;
- ❑ `Flags` — опции компиляции `D3DXSHADER`;
- ❑ `ppShader` — возвращает буфер, содержащий созданный шейдер. Этот буфер содержит откомпилированный код шейдера;
- ❑ `ppErrorMsgs` — возвращает листинг ошибок и предупреждений, с которыми столкнулись в течение компиляции;
- ❑ `ppConstantTable` — этот параметр возвращает указатель блоку `D3DXSHADER_CONSTANTTABLE`, который вставлен в шейдер.

D3DXCompileShaderFromFile()

Компилирует файл шейдера. Функция описывается таким образом:

```
HRESULT WINAPI D3DXCompileShaderFromFile(
    LPCTSTR                pSrcFile,
    CONST D3DXMACRO*       pDefines,
    LPD3DXINCLUDE          pInclude,
    LPCTSTR                pFunctionName,
    LPCTSTR                pTarget,
```

```

        DWORD                Flags,
        LPD3DXBUFFER*        ppShader,
        LPD3DXBUFFER*        ppErrorMsgs,
        LPD3DXSHADER_CONSTANTTABLE* ppConstantTable
    );

```

Параметры:

- `pSrcFile` — указатель, содержащий имя файла;
- `pDefines` — указатель на препроцессор;
- `pInclude` — дополнительный указатель интерфейса `ID3DXInclude`;
- `pFunctionName` — указатель на входную функцию;
- `pTarget` — указатель на строку, содержащую цель компиляции;
- `Flags` — опции компиляции `D3DXSHADER`;
- `ppShader` — возвращает буфер, содержащий созданный шейдер. Этот буфер содержит откомпилированный код шейдера;
- `ppErrorMsgs` — возвращает листинг ошибок и предупреждений, с которыми столкнулись в течение компиляции;
- `ppConstantTable` — этот параметр возвращает указатель блоку `D3DXSHADER_CONSTANTTABLE`, который вставлен в шейдер.

D3DXCompileShaderFromResource()

Компилирует файл шейдера из ресурса. Функция описывается таким образом:

```

HRESULT WINAPI D3DXCompileShaderFromResource(
    HMODULE                hSrcModule,
    LPCTSTR                pSrcResource,
    CONST D3DXMACRO*       pDefines,
    LPD3DXINCLUDE          pInclude,
    LPCTSTR                pFunctionName,
    LPCTSTR                pTarget,
    DWORD                 Flags,
    LPD3DXBUFFER*          ppShader,
    LPD3DXBUFFER*          ppErrorMsgs,
    LPD3DXSHADER_CONSTANTTABLE* ppConstantTable
);

```

Параметры:

- `hSrcModule` — значение `NULL` устанавливает использование текущего модуля;

- ❑ `pSrcResource` — указатель, определяющий имя ресурса;
- ❑ `pDefines` — указатель на препроцессор;
- ❑ `pInclude` — дополнительный указатель интерфейса `ID3DXInclude`;
- ❑ `pFunctionName` — указатель, содержащий имя входной функции;
- ❑ `pTarget` — указатель на строку, которая содержит цель компиляции;
- ❑ `Flags` — опции компиляции `D3DXSHADER`;
- ❑ `ppShader` — возвращает буфер, содержащий созданный шейдер. Этот буфер содержит откомпилированный код шейдера;
- ❑ `ppErrorMsgs` — возвращает листинг ошибок и предупреждений, с которыми столкнулись в течение компиляции;
- ❑ `ppConstantTable` — этот параметр возвращает указатель блоку `D3DXSHADER_CONSTANTTABLE`, который вставлен в шейдер.

D3DXComputeNormalMap()

Преобразовывает карту высоты в нормальную карту. Функция описывается таким образом:

```
HRESULT D3DXComputeNormalMap(
    LPDIRECT3DTEXTURE9 pTexture,
    LPDIRECT3DTEXTURE9 pSrcTexture,
    const PALETTEENTRY* pSrcPalette,
    DWORD               Flags,
    DWORD               Channel,
    FLOAT               Amplitude
);
```

Параметры:

- ❑ `pTexture` — указатель на интерфейс `IDirect3DTexture9`, представляющий текстуру расположения;
- ❑ `pSrcTexture` — указатель на интерфейс `IDirect3DTexture9`, представляет исходную высоту карты текстуры;
- ❑ `pSrcPalette` — указатель на тип `PALETTEENTRY`, который содержит исходную палитру из 256 цветов или значение `NULL`;
- ❑ `Flags` — содержит один или несколько флагов `D3DX_NORMALMAP`, управляющих установкой карты;
- ❑ `Channel` — имеет один флаг `D3DX_CHANNEL`, определяющий источник информации о высоте;
- ❑ `Amplitude` — амплитуда.

D3DXCreateCubeTexture()

Создает пустую кубическую текстуру. Функция описывается таким образом:

```
HRESULT D3DXCreateCubeTexture(  
    LPDIRECT3DDEVICE9      pDevice,  
    UINT                   Size,  
    UINT                   MipLevels,  
    DWORD                  Usage,  
    D3DFORMAT              Format,  
    D3DPOOL                Pool,  
    LPDIRECT3DCUBETEXTURE9* ppCubeTexture  
);
```

Параметры:

- ❑ `pDevice` — указатель на интерфейс `IDirect3DDevice9`;
- ❑ `Size` — ширина и высота кубической текстуры в пикселах;
- ❑ `MipLevels` — количество слоев текстуры (mip levels);
- ❑ `Usage` — использование поверхности;
- ❑ `Format` — элемент `D3DFORMAT`, описывает запрошенный формат пиксела кубической текстуры;
- ❑ `Pool` — член `D3DPOOL`, описывает класс памяти;
- ❑ `ppCubeTexture` — адрес указателя на интерфейс `IDirect3DCubeTexture9`.

D3DXCreateCubeTextureFromFile()

Создает кубическую текстуру из файла. Функция описывается таким образом:

```
HRESULT D3DXCreateCubeTextureFromFile(  
    LPDIRECT3DDEVICE9      pDevice,  
    LPCTSTR                pSrcFile,  
    LPDIRECT3DCUBETEXTURE9* ppCubeTexture  
);
```

Параметры:

- ❑ `pDevice` — указатель на интерфейс `IDirect3DDevice9`;
- ❑ `pSrcFile` — указатель на имя файла;
- ❑ `ppCubeTexture` — адрес указатель на интерфейс `IDirect3DCubeTexture9`, представляющий собой созданный кубический объект текстуры.

D3DXCreateFont()

Создает шрифт. Функция описывается таким образом:

```
HRESULT D3DXCreateFont(
    LPDIRECT3DDEVICE9 pDevice,
    HFONT              hFont,
    LPD3DXFONT**       ppFont);
```

Параметры:

- ❑ `pDevice` — указатель на устройство Direct3D;
- ❑ `hFont` — переменная структуры `HFONT`, содержащая созданный ранее шрифт;
- ❑ `ppFont` — адрес указателя, содержащего результат инициализации шрифта.

D3DXCreateMeshFVF()

Создает мэш, использующий гибкий формат вершины (FVF). Функция описывается таким образом:

```
HRESULT D3DXCreateMeshFVF(
    DWORD NumFaces,
    DWORD NumVertices,
    DWORD Options,
    DWORD FVF,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXMESH*       ppMesh
);
```

Параметры:

- ❑ `NumFaces` — количество лицевых частей в мэше;
- ❑ `NumVertices` — количество граней мэша. Этот параметр должен быть больше, чем 0;
- ❑ `Options` — один или несколько флагов из перечисления `D3DXMESH`, определяющего вариант создания мэша;
- ❑ `FVF` — комбинация `D3DFVF`, которая описывает формат вершины мэша;
- ❑ `pDevice` — указатель на интерфейс `IDirect3DDevice9`;
- ❑ `ppMesh` — адрес указателя на интерфейс `ID3DXMesh`.

D3DXCreateSPMesh()

Создает упрощенный меш. Функция описывается таким образом:

```
HRESULT D3DXCreateSPMesh(  
    LPD3DXMESH                pMesh,  
    CONST DWORD*              pAdjacency,  
    CONST LPD3DXATTRIBUTEWEIGHTS pVertexAttributeWeights,  
    CONST FLOAT*              pVertexWeights,  
    LPD3DXSPMESH*             ppSMesh  
);
```

Параметры:

- `pMesh` — указатель на интерфейс `ID3DXMesh`, представляющий меш;
- `pAdjacency` — указатель на массив значений трех лицевых граней;
- `pVertexAttributeWeights` — указатель структуры `D3DXATTRIBUTEWEIGHTS`, содержащей атрибуты для каждого компонента вершины;
- `pVertexWeights` — указатель на массив вершин;
- `ppSMesh` — адрес указателя на интерфейс `ID3DXSPMesh`, представляющего меш.

D3DXCreateTextureFromFile()

Загружает текстуру из файла. Функция описывается таким образом:

```
HRESULT D3DXCreateTextureFromFile(  
    LPDIRECT3DDEVICE9 pDevice,  
    LPCTSTR           pSrcFile,  
    LPDIRECT3DTEXTURE9* ppTexture);
```

Параметры:

- `pDevice` — указатель на интерфейс `IDirect3DDevice9`, показывающий, к какому устройству присоединяется текстура;
- `pSrcFile` — имя файла загружаемой текстуры;
- `ppTexture` — адрес указателя на интерфейс `IDirect3DTexture9`. Является результатом всей операции.

D3DXCreateTextureFromFileEx()

Загружает текстуру из файла. Функция описывается таким образом:

```
HRESULT D3DXCreateTextureFromFileEx(  
    LPDIRECT3DDEVICE9 pDevice,  
    LPCTSTR           pSrcFile,
```

```

UINT           Width,
UINT           Height,
UINT           MipLevels,
DWORD          Usage,
D3DFORMAT      Format,
D3DPOOL        Pool,
DWORD          Filter,
DWORD          MipFilter,
D3DCOLOR       ColorKey,
D3DXIMAGE_INFO* pSrcInfo,
PALETTEENTRY*  pPalette,
LPDIRECT3DTEXTURE9* ppTexture

```

```
);
```

Параметры:

- `pDevice` — указатель на интерфейс `IDirect3DDevice9`;
- `pSrcFile` — указатель на строку, обозначающую имя файла загружаемой текстуры;
- `Width` — ширина текстуры в пикселах, но есть возможность установки нулевого значения или флага `D3DX_DEFAULT` для обработки значения по умолчанию;
- `Height` — высота текстуры в пикселах. Тоже можно выставить значение в 0 или применить флаг `D3DX_DEFAULT`;
- `MipLevels` — "мип-уровень", используется в "мип-маппинге";
- `Usage` — формат используемой поверхности;
- `Format` — член перечисляемого типа `D3DFORMAT`. Текстура может иметь различный формат, поэтому его необходимо задавать вручную либо воспользоваться флагом `D3DFMT_UNKNOWN` — значение по умолчанию. Этот флаг сам определит формат текстуры и передаст его в функцию;
- `Pool` — описывает класс памяти, в который данная текстура будет загружена. Член перечисляемого типа `D3DPOOL`;
- `Filter` — фильтр, определяющий фильтрацию текстуры. Здесь можно установить флаги управления из `D3DX_FILTER`. Значение по умолчанию `D3DX_DEFAULT`;
- `MipFilter` — фильтр, определяющий фильтрацию текстур в "мип-маппинге". Используются флаги управления из `D3DX_FILTER`, значение по умолчанию — `D3DX_DEFAULT`;
- `ColorKey` — это цветовой ключ, в котором определяется нерисуемый цвет. Значение `D3DCOLOR` всегда 32-битное ARGB, альфа-канал устанавливается в FF. Например, если вы хотите не прорисовывать черный цвет,

то значение должно быть равно 0xFF000000. Когда в этом параметре применяется значение, равное 0, то цветовой ключ будет отключен;

- `pSrcInfo` — указатель на структуру `D3DXIMAGE_INFO`. Эта структура заполняется данными об исходном файле текстуры. Значение `NULL` установлено по умолчанию;
- `pPalette` — указатель на структуру `PALETTEENTRY`, определяющую 256-цветную палитру. Для заполнения можно установить значение `NULL`;
- `ppTexture` — адрес указателя на интерфейс `IDirect3DTexture9`, результат всей операции.

D3DXDeclaratorFromFVF()

Возвращает декларацию формата вершины кода FVF. Функция описывается таким образом:

```
HRESULT D3DXDeclaratorFromFVF(  
    DWORD                FVF,  
    D3DVERTEXELEMENT9 Declaration[MAX_FVF_DECL_SIZE]  
);
```

Параметры:

- `FVF` — комбинация `D3DFVF`, с помощью которой описывается формат;
- `Declaration` — массив элементов `D3DVERTEXELEMENT9`, описывающих формат вершины.

D3DXGetImageInfoFromFile()

Извлекает информацию о файле образа. Функция описывается так:

```
HRESULT D3DXGetImageInfoFromFile(  
    LPCSTR                pSrcFile,  
    D3DXIMAGE_INFO* pSrcInfo  
);
```

Параметры:

- `pSrcFile` — файловое имя образа;
- `pSrcInfo` — указатель на структуру `D3DXIMAGE_INFO`, которую нужно заполнять описанием данных в исходном файле.

D3DXLoadMeshFromX()

Загружает меш из X-файла. Функция описывается таким образом:

```
HRESULT D3DXLoadMeshFromX(  
    LPCTSTR                pFilename,
```

```

    DWORD                Options,
    LPDIRECT3DDEVICE9    pDevice,
    LPD3DXBUFFER*        ppAdjacency,
    LPD3DXBUFFER*        ppMaterials,
    LPD3DXBUFFER*        ppEffectInstances,
    DWORD*               pNumMaterials,
    LPD3DXMESH*          ppMesh
);

```

Параметры:

- ❑ `pFilename` — указатель на строку, в которой указано имя загружаемого X-файла;
- ❑ `Options` — член перечисляемого типа `D3DXMESH`, использует один или несколько флагов при создании мэша;
- ❑ `pDevice` — это указатель на интерфейс `IDirect3DDevice9`, связывающий X-файл с устройством `Direct3D`;
- ❑ `ppAdjacency` — адрес указателя на буфер, содержащий данные о смежности. В этом параметре используется указатель на интерфейс `ID3DXBuffer`;
- ❑ `ppMaterials` — адрес указателя на интерфейс `ID3DXBuffer`, содержащий значения структуры `D3DXMATERIAL` для X-файла;
- ❑ `ppEffectInstances` — адрес указателя на интерфейс `ID3DXBuffer`, содержащий специфические эффекты;
- ❑ `pNumMaterials` — указатель на массив данных для структуры `D3DXMATERIAL`;
- ❑ `ppMesh` — адрес указателя на интерфейс `ID3DXMesh`, результат всей операции.

D3DXMatrixIdentity()

Создает матрицу тождества. Функция описывается таким образом:

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

Параметр:

- ❑ `pOut` — указатель на структуру `D3DXMATRIX`, являющуюся результатом операции.

D3DXMatrixInverse()

Вычисляет инверсию матрицы. Функция описывается таким образом:

```

D3DXMATRIX* D3DXMatrixInverse(
    D3DXMATRIX*    pOut,

```

```
    FLOAT*                pDeterminant,  
    CONST D3DXMATRIX*    pM  
);
```

Параметры:

- ❑ pOut — указатель на структуру D3DXMATRIX, являющуюся результатом операции;
- ❑ pDeterminant — указатель на величину FLOAT, содержащую детерминант матрицы;
- ❑ pM — указатель на исходную структуру D3DXMATRIX.

D3DXMatrixLookAtLH()

Представляет левостроннюю систему координат. Функция описывается таким образом:

```
D3DXMATRIX* D3DXMatrixLookAtLH(  
    D3DXMATRIX*        pOut,  
    CONST D3DXVECTOR3* pEye,  
    CONST D3DXVECTOR3* pAt,  
    CONST D3DXVECTOR3* pUp);
```

Параметры:

- ❑ pOut — результат выполнения этой функции. Указатель на структуру D3DXMATRIX;
- ❑ pEye — указатель на структуру D3DXVECTOR3, который определяет точку, откуда происходит просмотр сцены, т. е. место, где мы находимся;
- ❑ pAt — указатель на структуру D3DXVECTOR3, определяющий то, на что мы смотрим;
- ❑ pUp — указатель на структуру D3DXVECTOR3, определяющий то, что мы видим.

D3DMatrixMultiply()

Результат умножения двух матриц. Функция описывается таким образом:

```
D3DMATRIX* D3DMatrixMultiply(  
    D3DMATRIX*        pOut,  
    CONST D3DMATRIX*  pM1,  
    CONST D3DMATRIX*  pM2)
```

Параметры:

- ❑ pOut — указатель на структуру D3DMATRIX, который является результатом действий по умножению между собой двух матриц;

- pM1 — указатель на исходную структуру D3DXMATRIX;
- pM2 — указатель на исходную структуру D3DXMATRIX.

D3DXMatrixMultiplyTranspose()

Вычисляет преобразование от двух матриц. Функция описывается таким образом:

```
D3DXMATRIX* D3DXMatrixMultiplyTranspose(
    D3DXMATRIX*      pOut,
    const D3DXMATRIX* pM1,
    const D3DXMATRIX* pM2
);
```

Параметры:

- pOut — указатель на структуру D3DXMATRIX, являющийся результатом операции;
- pM1 — указатель на исходную структуру D3DXMATRIX;
- pM2 — указатель на исходную структуру D3DXMATRIX.

D3DXMatrixPerspectiveFovLH()

Представляет правостороннюю систему координат. Функция описывается таким образом:

```
D3DXMATRIX* D3DXMatrixPerspectiveFovLH (
    D3DXMATRIX* pOut,
    FLOAT        fovY,
    FLOAT        Aspect,
    FLOAT        zn,
    FLOAT        zf);
```

Параметры:

- pOut — результат всей операции, указатель на структуру D3DXMATRIX;
- fovY — определяет поле зрения в направлении оси *Y*, измеряется в радианах. Типичное поле — D3DX_PI/4;
- Aspect — коэффициент сжатия для соотношения геометрических размеров. Обычно значение равно единице;
- zn — передний план отсечения сцены;
- zf — задний план отсечения сцены. За границами двух параметров zn и zf ничего не рисуется.

D3DXMatrixRotationX()

Вращение по оси *X*. Функция описывается таким образом:

```
D3DXMATRIX* D3DXMatrixRotationX(  
    D3DXMATRIX* pOut,  
    FLOAT       Angle);
```

Параметры:

- *pOut* — указатель на структуру *D3DXMATRIX*, в которую помещается исходный результат операции;
- *Angle* — угол вращения в радианах по оси *X*.

D3DXMatrixRotationY()

Вращение по оси *Y*. Функция описывается таким образом:

```
D3DXMATRIX* D3DXMatrixRotationY(  
    D3DXMATRIX *pOut,  
    FLOAT       Angle);
```

Параметры:

- *pOut* — указатель на структуру *D3DXMATRIX*, в которую помещается исходный результат операции;
- *Angle* — угол вращения в радианах по оси *Y*.

D3DXMatrixRotationZ()

Вращение по оси *Z*. Функция описывается таким образом:

```
D3DXMATRIX* D3DXMatrixRotationZ(  
    D3DXMATRIX* pOut,  
    FLOAT       Angle);
```

Параметры:

- *pOut* — указатель на структуру *D3DXMATRIX*, в которую помещается исходный результат операции;
- *Angle* — угол вращения в радианах по оси *Z*.

D3DXMatrixScaling()

Строит матрицу на основании коэффициента сжатия или растяжения. Функция описывается таким образом:

```
D3DXMATRIX* D3DXMatrixScaling(  
    D3DXMATRIX* pOut,  
    FLOAT       sx,
```

```

    FLOAT    sy,
    FLOAT    sz
);

```

Параметры:

- `pOut` — указатель на структуру `D3DXMATRIX`, являющуюся результатом операции;
- `sx` — коэффициент масштабирования по оси X ;
- `sy` — коэффициент масштабирования по оси Y ;
- `sz` — коэффициент масштабирования по оси Z .

D3DXMatrixTranslation()

Строит матрицу, используя коэффициент трансформации. Функция описывается таким образом:

```

D3DXMATRIX* D3DXMatrixTranslation(
    D3DXMATRIX* pOut,
    FLOAT      x,
    FLOAT      y,
    FLOAT      z
);

```

Параметры:

- `pOut` — указатель на структуру `D3DXMATRIX`, являющуюся результатом операции;
- `x` — координата по оси X ;
- `y` — координата по оси Y ;
- `z` — координата по оси Z .

D3DXMatrixTranspose()

Возвращает матрицу трансформации. Функция описывается таким образом:

```

D3DXMATRIX* D3DXMatrixTranspose(
    D3DXMATRIX* pOut,
    CONST D3DXMATRIX* pM
);

```

Параметры:

- `pOut` — указатель на структуру `D3DXMATRIX`, являющуюся результатом операции;
- `pM` — указатель на исходную структуру `D3DXMATRIX`.

D3DXSaveTextureToFile()

Сохраняет текстуру в файл. Функция описывается таким образом:

```
HRESULT D3DXSaveTextureToFile(
    LPCTSTR                pDestFile,
    D3DXIMAGE_FILEFORMAT   DestFormat,
    LPDIRECT3DBASETEXTURE9 pSrcTexture,
    const PALETTEENTRY*     pSrcPalette
);
```

Параметры:

- `pDestFile` — имя файла;
- `DestFormat` — `D3DXIMAGE_FILEFORMAT` — флаг, определяющий файловый формат;
- `pSrcTexture` — указатель на интерфейс `IDirect3DBaseTexture9`, содержащий текстуру;
- `pSrcPalette` — указатель на структуру `PALETTEENTRY`, содержащую палитру из 256 цветов. Этот параметр может быть `NULL`.

D3DXVec3Normalize()

Возвращает нормализованную версию 3D-вектора. Функция описывается таким образом:

```
D3DXVECTOR3* D3DXVec3Normalize(
    D3DXVECTOR3*    pOut;
    CONST D3DXVECTOR* pV);
```

Параметры:

- `pOut` — указатель на структуру `D3DXVECTOR3`, выходной результат всей операции;
- `pV` — указатель на структуру `D3DXVECTOR`, исходное значение.

Структуры DirectX Graphics

Этот раздел представляет набор различных структур из интерфейса DirectX Graphics.

D3DADAPTER_IDENTIFIER9

Содержит идентифицирующую информацию об адаптере. Структура описывается таким образом:

```
typedef struct _D3DADAPTER_IDENTIFIER9 {
    char                Driver[MAX_DEVICE_IDENTIFIER_STRING];
```

```

char          Description[MAX_DEVICE_IDENTIFIER_STRING];
char          DeviceName[32];
LARGE_INTEGER DriverVersion;
DWORD         DriverVersionLowPart;
DWORD         DriverVersionHighPart;
DWORD         VendorId;
DWORD         DeviceId;
DWORD         SubSysId;
DWORD         Revision;
GUID          DeviceIdentifier;
DWORD         WHQLLevel;
} D3DADAPTER_IDENTIFIER9;

```

Параметры:

- ❑ **Driver** — информация этого параметра используется для представления пользователю;
- ❑ **Description** — информация этого параметра используется для представления пользователю;
- ❑ **DeviceName** — имя установленного устройства;
- ❑ **DriverVersion** — идентифицирует версию драйвера;
- ❑ **DriverVersionLowPart** — идентифицирует версию драйвера Direct3D;
- ❑ **DriverVersionHighPart** — идентифицирует версию драйвера Direct3D;
- ❑ **VendorId** — может быть использовано для идентификации установленного чипа на видеоадаптере;
- ❑ **DeviceId** — может быть использовано для идентификации установленного чипа на видеоадаптере;
- ❑ **SubSysId** — может быть использовано для идентификации системы;
- ❑ **Revision** — может быть использовано для идентификации ревизии;
- ❑ **DeviceIdentifier** — идентифицирует устройство;
- ❑ **WHQLLevel** — Windows Hardware Quality Labs (WHQL).

D3DBOX

Определяет объем. Структура описывается таким образом:

```

typedef struct _D3DBOX {
    UINT Left;
    UINT Top;
    UINT Right;

```

```
UINT Bottom;  
UINT Front;  
UINT Back;  
} D3DBOX;
```

Параметры:

- ☐ Left — левая сторона куба;
- ☐ Top — верх куба;
- ☐ Right — правая сторона куба;
- ☐ Bottom — низ куба;
- ☐ Front — внешняя сторона куба;
- ☐ Back — обратная сторона куба.

D3DCOLORVALUE

Описывает цветовые величины. Структура описывается таким образом:

```
typedef struct _D3DCOLORVALUE {  
    float r;  
    float g;  
    float b;  
    float a;  
} D3DCOLORVALUE;
```

Параметры:

- ☐ r — величина с плавающей точкой, определяющая красный компонент цвета. Принимает значения в диапазоне от 0.0 до 1.0, где 0.0 — черный цвет;
- ☐ g — величина с плавающей точкой, определяющая зеленый компонент цвета. Принимает значения в диапазоне от 0.0 до 1.0, где 0.0 — черный цвет;
- ☐ b — величина с плавающей точкой, определяющая синий компонент цвета. Принимает значения в диапазоне от 0.0 до 1.0, где 0.0 — черный цвет;
- ☐ a — величина с плавающей точкой, определяющая альфа-компонент цвета. Принимает значения в диапазоне от 0.0 до 1.0, где 0.0 — черный цвет.

D3DDISPLAYMODE

Описывает параметры дисплея. Структура описывается таким образом:

```
typedef struct _D3DDISPLAYMODE {  
    UINT        Width;
```

```

    UINT      Height;
    UINT      RefreshRate;
    D3DFORMAT Format;
} DISPLAYMODE;

```

Параметры:

- ❑ `Width` — ширина рабочей поверхности экрана в пикселах;
- ❑ `Height` — высота рабочей поверхности экрана в пикселах;
- ❑ `RefreshRate` — частота регенерации. Значение 0 указывает на значение по умолчанию;
- ❑ `Format` — член перечисляемого типа `D3DFORMAT`, описывающий формат поверхности дисплея.

D3DLIGHT9

Определяет свойства света. Структура описывается таким образом:

```

typedef struct _D3DLIGHT9 {
    D3DLIGHTTYPE  Type;
    D3DCOLORVALUE Diffuse;
    D3DCOLORVALUE Specular;
    D3DCOLORVALUE Ambient;
    D3DVECTOR      Position;
    D3DVECTOR      Direction;
    FLOAT          Range;
    FLOAT          Falloff;
    FLOAT          Attenuation_0;
    FLOAT          Attenuation_1;
    FLOAT          Attenuation_2;
    FLOAT          Theta;
    FLOAT          Phi;
} D3DLIGHT9;

```

Параметры:

- ❑ `Type` — член перечисляемого типа `D3DLIGHTTYPE`. Это значение определяет тип источника света;
- ❑ `Diffuse` — диффузный или рассеянный свет, излучаемый источником света. Член структуры `D3DCOLORVALUE`;
- ❑ `Specular` — зеркальный свет, излучаемый источником света. Член структуры `D3DCOLORVALUE`;

- ❑ **Ambient** — окружающий свет, излучаемый источником света. Член структуры `D3DCOLORVALUE`;
- ❑ **Position** — в этом параметре с помощью структуры `D3DVECTOR` задаются координаты источника света в сцене. Когда используется направленный или параллельный (*directional*) источник света, этот параметр игнорируется, поскольку этот тип не имеет определенного источника света;
- ❑ **Direction** — задается структурой `D3DVECTOR` и определяет направление падающего света. Работает только с направленным (*directional*) и прожекторным (*spotlight*) типом источника света;
- ❑ **Range** — максимальное расстояние для освещения выбранного источника света;
- ❑ **Falloff** — этот параметр уменьшает освещение между внутренним конусом, определенным как угол *Theta* и внешним углом *Phi* источника света. Например, как в фонарике — большой круг света, внутри которого меньший круг от лампы более ярко светящийся. Значение по умолчанию 1.0, для более равномерного перехода от двух конусов;
- ❑ **Attenuation_0**, **Attenuation_1** и **Attenuation_2** — эти три параметра влияют на интенсивность освещения за счет изменения расстояния от объекта до источника света;
- ❑ **Theta** — угол внутреннего конуса источника света, измеряется в радианах. Минимальное значение 0, максимальное определено в параметре *Phi*;
- ❑ **Phi** — угол внешнего конуса источника света, измеряется в радианах. Минимальное значение 0, максимальное — константа π . За границами этого конуса освещение не происходит.

D3DMATERIAL9

Устанавливает свойства материала. Структура описывается таким образом:

```
tupedef struct _D3DMATERIAL9 {  
    D3DCOLORVALUE Diffuse;  
    D3DCOLORVALUE Ambient;  
    D3DCOLORVALUE Specular;  
    D3DCOLORVALUE Emissive;  
    FLOAT          Power;  
} D3DMATERIAL9;
```

Параметры:

- ❑ **Diffuse** — значение структуры `D3DCOLORVALUE`, определяет диффузный или рассеянный свет материала;
- ❑ **Ambient** — значение структуры `D3DCOLORVALUE`, определяет цвет подсветки, иногда еще говорят — окружающий свет;

- ❑ **Specular** — значение структуры `D3DCOLORVALUE`, определяет отражающий или зеркальный свет. С его помощью можно определить вид бликов;
- ❑ **Emissive** — значение структуры `D3DCOLORVALUE`, определяет эмиссионный или излучательный свет материала;
- ❑ **Power** — это размер, а скорее мощность отражения, определенная для материала. Когда необходимо отключить блики, значение выставляется в 0.

D3DPRESENT_PARAMETERS

Описывает параметры представления устройства. Структура описывается таким образом:

```
typedef struct _D3DPRESENT_PARAMETERS{
    UINT                BackBufferWidth;
    UINT                BackBufferHeight;
    D3DFORMAT           BackBufferFormat;
    UINT                BackBufferCount;
    D3DMULTISAMPLE_TYPE MultiSampleType;
    D3DSWAPEFFECT        SwapEffect;
    HWND                hDeviceWindow;
    BOOL                Windowed;
    BOOL                EnableAutoDepthStencil;
    D3DFORMAT           AutoDepthStencilFormat;
    DWORD               Flags;
    UINT                FullScreen_RefreshRateInHz;
    UINT                FullScreen_PresentationInterval;
} D3DPRESENT_PARAMETERS;
```

Параметры:

- ❑ **BackBufferWidth** — ширина заднего буфера в пикселах;
- ❑ **BackBufferHeight** — высота заднего буфера в пикселах;
- ❑ **BackBufferFormat** — формат поверхности заднего буфера. Задний буфер должен соответствовать текущим настройкам видеорежима;
- ❑ **BackBufferCount** — это число задних буферов. Значение может быть 0, 1, 2 или 3. Один задний буфер считается минимальным;
- ❑ **MultiSampleType** — перечисляемый тип, член `D3DMULTISAMPLE_TYPE`. Должен быть определен как `D3DMULTISAMPLE_NONE`, если параметр `SwapEffect` не был установлен в значение `D3DSWAPEFFECT_DISCARD`;
- ❑ **SwapEffect** — этот параметр служит для определения обмена буферов. Перечисляемый тип, член `D3DSWAPEFFECT`. Так, если значение `Windowed` —

TRUE и параметр `SwapEffect` установлен в `D3DSWAPEFFECT_FLIP`, то создается один задний буфер;

- ❑ `hDeviceWindow` — если приложение работает в полноэкранном режиме, то задействуется вся поверхность экрана монитора. Если параметр `Windowed` имеет значение TRUE, то это оконный режим, если же устанавливается значение FALSE, то приложение будет работать в полноэкранном режиме;
- ❑ `Windowed` — это видеорежим, используемый вашим приложением. Значение TRUE определяет оконный режим. FALSE — это полноэкранный режим, т. е. во весь экран;
- ❑ `EnableAutoDepthStencil` — если этот параметр будет иметь значение TRUE, то для Direct3D 9 становится доступен буфер глубины окна, и устройство Direct3D 9 создаст буфер трафарета;
- ❑ `AutoDepthStencilFormat` — член перечисляемого типа `D3DFORMAT`. Формат автоматической поверхности трафарета созданного устройства. Этот параметр игнорируется, если `EnableAutoDepthStencil` не установлен в значение TRUE;
- ❑ `Flags` — этот параметр может быть установлен в 0, или применен флаг `D3DPRESENTFLAG_LOCKABLE_BACKBUFFER`. Этот флаг устанавливается тогда, когда приложение требует блокировать задний буфер;
- ❑ `FullScreen_RefreshRateInHz` — частота обновления экрана;
- ❑ `FullScreen_PresentationInterval` — максимальный интервал переключения заднего буфера. Для полноэкранного переключения может использоваться `D3DPRESENT_INTERVAL_DEFAULT`, или значение, соответствующее одному из флагов, перечисленных в структуре `D3DCAPS9`.

D3DRANGE

Предопределяет диапазон. Структура описывается таким образом:

```
typedef struct _D3DRANGE {  
    UINT Offset;  
    UINT Size;  
} D3DRANGE;
```

Параметры:

- ❑ `Offset` — компенсирует заданный диапазон в байтах;
- ❑ `Size` — размер в байтах.

D3DRECT

Определяет прямоугольную область. Структура описывается таким образом:

```
typedef struct _D3DRECT {
    LONG x1;
    LONG y1;
    LONG x2;
    LONG y2;
} D3DRECT
```

Параметры:

- $x1$ и $y1$ — координаты левого верхнего угла прямоугольника;
- $x2$ и $y2$ — координаты правого нижнего угла прямоугольника

D3DVECTOR

Определяет вектор. Структура описывается таким образом:

```
typedef struct _D3DVECTOR {float x, y, z;}
D3DVECTOR;
```

Параметры:

- x, y, z — значения с плавающей точкой, описывают вектор.

D3DVERTEXELEMENT9

Определяет входные вершинные данные в конвейере. Структура описывается таким образом:

```
typedef struct _D3DVERTEXELEMENT9 {
    BYTE Stream;
    BYTE Offset;
    BYTE Type;
    BYTE Method;
    BYTE Usage;
    BYTE UsageIndex;
} D3DVERTEXELEMENT9;
```

Параметры:

- *Stream* — номер потока;
- *Offset* — первый такт для чтения;
- *Type* — тип входных данных;
- *Method* — операторы. Имеется несколько операторов, по умолчанию `D3DDECLMETHOD_DEFAULT`;

- Usage — определяет используемые данные, например, вершины, нормали, текстуры;
- UsageIndex — специфический параметр, определяет использование предполагаемых данных.

D3DVIEWPORT9

Определяет поверхность для рендеринга. Структура описывается таким образом:

```
typedef struct _D3DVIEWPORT9 {  
    DWORD X;  
    DWORD Y;  
    DWORD Width;  
    DWORD Height;  
    Float MinZ;  
    Float MaxZ;  
} D3DVIEWPORT9;
```

Параметры:

- X — пиксельная координата X верхнего левого угла. 0 для всей поверхности;
- Y — пиксельная координата Y верхнего левого угла. 0 для всей поверхности;
- Width — ширина поверхности в пикселах;
- Height — высота поверхности в пикселах;
- MinZ — минимальный диапазон глубины. Обычно это значение равно 0.0;
- MaxZ — максимальный диапазон глубины. Обычно это значение равно 1.0.

D3DXMATERIAL

Возвращает информацию о материале, используется при работе с X-файлами. Структура описывается таким образом:

```
tupedef struct D3DXMATERIAL{  
    D3DMATERIAL9 MatD3D;  
    LPSTR pTextureFilename;  
} D3DXMATERIAL;
```

Параметры:

- MatD3D — структура D3DMATERIAL9, описывающая материальные свойства;
- pTextureFilename — указатель на строку, определяющую имя файла текстуры.

RECT

Определяет прямоугольную область. Структура описывается таким образом:

```
typedef struct _RECT {
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
} RECT, *PRECT;
```

Параметры:

- `left` — координата по оси *X* левого верхнего угла прямоугольника;
- `top` — координата по оси *Y* левого верхнего угла прямоугольника;
- `right` — координата по оси *X* правого нижнего угла прямоугольника;
- `bottom` — координата по оси *Y* правого нижнего угла прямоугольника.

Типы DirectX Graphics

Перечисляемые типы данных, содержащиеся в интерфейсе DirectX Graphics.

D3DBLEND

Определяет поддерживаемые способы смешивания. Тип описывается таким образом:

```
typedef enum _D3DBLEND {
    D3DBLEND_ZERO           = 1,
    D3DBLEND_ONE            = 2,
    D3DBLEND_SRCCOLOR       = 3,
    D3DBLEND_INVSRCCOLOR    = 4,
    D3DBLEND_SRCALPHA        = 5,
    D3DBLEND_INVSRCALPHA    = 6,
    D3DBLEND_DESTALPHA      = 7,
    D3DBLEND_INVDESTALPHA   = 8,
    D3DBLEND_DESTCOLOR      = 9,
    D3DBLEND_INVDESTCOLOR   = 10,
    D3DBLEND_SRCALPHASAT    = 11,
    D3DBLEND_BOTHSRCALPHA   = 12,
    D3DBLEND_BOTHINVSRCALPHA = 13,
    D3DBLEND_BLENDFACTOR    = 14,
```

```

D3DBLEND_INVBLENDFACTOR = 15,
D3DBLEND_FORCE_DWORD    = 0x7fffffff
} D3DBLEND;

```

Параметры:

- ❑ D3DBLEND_ZERO — показатель смешивания. Задается значениями (0, 0, 0, 0);
- ❑ D3DBLEND_ONE — показатель смешивания. Задается значениями (1, 1, 1, 1);
- ❑ D3DBLEND_SRCCOLOR — показатель смешивания. Задается значениями (Rs, Gs, Bs, As);
- ❑ D3DBLEND_INVSRCOLOR — показатель смешивания. Определяется такими значениями (1 – Rs, 1 – Gs, 1 – Bs, 1 – As);
- ❑ D3DBLEND_SRCALPHA — показатель смешивания (As, As, As, As);
- ❑ D3DBLEND_INVSRCALPHA — показатель смешивания. Определяется следующими значениями (1 – As, 1 – As, 1 – As, 1 – As);
- ❑ D3DBLEND_DESTALPHA — показатель смешивания (Ad, Ad, Ad, Ad);
- ❑ D3DBLEND_INVDESTALPHA — показатель смешивания. Определяется следующими значениями (1 – Ad, 1 – Ad, 1 – Ad, 1 – Ad);
- ❑ D3DBLEND_DESTCOLOR — показатель смешивания (Rd, Gd, Bd, Ad);
- ❑ D3DBLEND_INVDESTCOLOR — показатель смешивания. Определяется следующими значениями (1 – Rd, 1 – Gd, 1 – Bd, 1 – Ad);
- ❑ D3DBLEND_SRCALPHASAT — показатель смешивания (f, f, f, 1), $f = \min(Ad, 1 - Ad)$;
- ❑ D3DBLEND_BOTHSRCALPHA — устаревший параметр, больше не используется;
- ❑ D3DBLEND_BOTHINVSRCALPHA — исходный показатель смешивания. Определяется значениями (1 – As, 1 – As, 1 – As, 1 – As);
- ❑ D3DBLEND_BLENDFACTOR — постоянный показатель смешивания. Этот способ смешивания поддерживается, только если установлен параметр D3DPBLEND_BLENDFACTOR;
- ❑ D3DBLEND_INVBLENDFACTOR — инвертированный показатель смешивания. Этот способ смешивания поддерживается, только если параметр D3DPBLEND_BLENDFACTOR установлен в SrcBlendCaps;
- ❑ D3DBLEND_FORCE_DWORD — этот параметр не используется.

D3DBLENDOP

Определяет поддерживаемые действия смешивания. Тип описывается таким образом:

```

typedef enum _D3DBLENDOP {
    D3DBLENDOP_ADD          = 1,

```

```

D3DBLENDOP_SUBTRACT      = 2,
D3DBLENDOP_REVSUBTRACT   = 3,
D3DBLENDOP_MIN           = 4,
D3DBLENDOP_MAX           = 5,
D3DBLENDOP_FORCE_DWORD   = 0x7fffffff
} D3DBLENDOP;

```

Параметры:

- ❑ D3DBLENDOP_ADD — результат, определяемый по следующей формуле:
Result = Source + Destination;
- ❑ D3DBLENDOP_SUBTRACT — результат, определяемый по следующей формуле:
Result = Source - Destination;
- ❑ D3DBLENDOP_REVSUBTRACT — результат, определяемый по следующей формуле:
Result = Destination - Source;
- ❑ D3DBLENDOP_MIN — результат, определяемый по следующей формуле:
Result = min(Source, Destination);
- ❑ D3DBLENDOP_MAX — результат, определяемый по следующей формуле:
Result = max(Source, Destination);
- ❑ D3DBLENDOP_FORCE_DWORD — этот параметр не используется.

D3DCULL

Определяет тип отсечения. Тип описывается таким образом:

```

typedef enum _D3DCULL {
    D3DCULL_NONE      = 1;
    D3DCULL_CW        = 2;
    D3DCULL_CCW        = 3;
    D3DCULL_FORCE_DWORD = 0x7fffffff
} D3DCULL;

```

Параметры:

- ❑ D3DCULL_NONE — отсечение задних граней отключается;
- ❑ D3DCULL_CW — отсечение включено и происходит по часовой стрелке;
- ❑ D3DCULL_CCW — отсечение включено и происходит против часовой стрелки;
- ❑ D3DCULL_FORCE_DWORD — значение не используется.

D3DDEVTYPE

Определяет тип устройства. Тип описывается таким образом:

```

typedef enum _D3DDEVTYPE {
    D3DDEVTYPE_HAL      = 1,

```

```
D3DDEVTYPE_REF          = 2,  
D3DDEVTYPE_SW           = 3,  
D3DDEVTYPE_FORCE_DWORD = 0xffffffff  
} D3DDEVTYPE;
```

Параметры:

- ❑ D3DDEVTYPE_HAL — используются возможности аппаратного обеспечения;
- ❑ D3DDEVTYPE_REF — эмулирует программно;
- ❑ D3DDEVTYPE_SW — это значение применяется вместе с функцией `IDirect3D9::RegisterSoftwareDevice`;
- ❑ D3DDEVTYPE_FORCE_DWORD — параметр не используется.

D3DLIGHTTYPE

Определяет тип используемого источника освещения. Тип описывается таким образом:

```
typedef enum _D3DLIGHTTYPE {  
    D3DLIGHT_POINT          = 1,  
    D3DLIGHT_SPOT           = 2,  
    D3DLIGHT_DIRECTIONAL    = 3,  
    D3DLIGHT_FORCE_DWORD    = 0x7fffffff,  
} D3DLIGHTTYPE;
```

Параметры:

- ❑ D3DLIGHT_POINT — точечный источник света, рассеивающий свет во все стороны. Этот источник рассчитывается немного медленнее, но отображается лучше;
- ❑ D3DLIGHT_SPOT — прожекторный или нацеленный источник света, светящийся в определенном направлении в виде конуса;
- ❑ D3DLIGHT_DIRECTIONAL — параллельный или направленный свет, не имеющий определенного источника. Может распространяться до бесконечности;
- ❑ D3DLIGHT_FORCE_DWORD — параметр не используется.

D3DPOLL

Определяет класс памяти ресурса. Тип описывается таким образом:

```
typedef enum _D3DPOLL {  
    D3DPOOL_DEFAULT          = 0,  
    D3DPOOL_MANAGED          = 1,  
    D3DPOOL_SYSTEMMEM        = 2,  
}
```

```

D3DPOOL_SCRATCH      = 3,
D3DPOOL_FORCE_DWORD = 0x7fffffff
} D3DPOOL;

```

Параметры:

- ☐ D3DPOOL_DEFAULT — с помощью этой константы ресурсы помещаются в память, объединяя самые лучшие способы использования этого ресурса;
- ☐ D3DPOOL_MANAGED — ресурсы автоматически копируются в доступную для устройства память;
- ☐ D3DPOOL_SYSTEMMEM — использует системную оперативную память;
- ☐ D3DPOOL_SCRATCH — ресурс помещается в системную память и не обновляется при потере устройства;
- ☐ D3DPOOL_FORCE_DWORD — значение не используется.

D3DPRIMITIVETYPE

Определяет примитивы, поддерживаемые Direct3D. Тип описывается таким образом:

```

typedef enum _D3DPRIMITIVETYPE {
    D3DPT_POINTLIST      = 1,
    D3DPT_LINELIST       = 2,
    D3DPT_LINESTRIP      = 3,
    D3DPT_TRIANGLELIST   = 4,
    D3DPT_TRIANGLESTRIP  = 5,
    D3DPT_TRIANGLEFAN    = 6,
    D3DPT_FORCE_DWORD    = 0x7fffffff
} D3DPRIMITIVETYPE;

```

Параметры:

- ☐ D3DPT_POINTLIST — рендеринг изолированных точек;
- ☐ D3DPT_LINELIST — рендеринг изолированных линий;
- ☐ D3DPT_LINESTRIP — рендеринг полосы связанных линий;
- ☐ D3DPT_TRIANGLELIST — рендеринг изолированных треугольников;
- ☐ D3DPT_TRIANGLESTRIP — рендеринг полосы связанных треугольников;
- ☐ D3DPT_TRIANGLEFAN — рендеринг треугольников, расположенных в виде вентилятора. Это когда одна из вершин каждого треугольника стыкуется в центре, создавая таким образом окружность, похожую на вентилятор;
- ☐ D3DPT_FORCE_DWORD — этот параметр не используется.

D3DRENDERSTATETYPE

Определяет состояния устройства рендеринга. Тип описывается таким образом:

```
typedef enum _D3DRENDERSTATETYPE {
    D3DRS_ZENABLE                      = 7,
    D3DRS_FILLMODE                     = 8,
    D3DRS_SHADEMODE                     = 9,
    D3DRS_ZWRITEENABLE                 = 14,
    D3DRS_ALPHATESTENABLE              = 15,
    D3DRS_LASTPIXEL                    = 16,
    D3DRS_SRCBLEND                      = 19,
    D3DRS_DESTBLEND                    = 20,
    D3DRS_CULLMODE                     = 22,
    D3DRS_ZFUNC                        = 23,
    D3DRS_ALPHAREF                     = 24,
    D3DRS_ALPHAFUNC                    = 25,
    D3DRS_DITHERENABLE                 = 26,
    D3DRS_ALPHABLENDENABLE             = 27,
    D3DRS_FOGENABLE                    = 28,
    D3DRS_SPECULARENABLE               = 29,
    D3DRS_FOGCOLOR                     = 34,
    D3DRS_FOGTABLEMODE                 = 35,
    D3DRS_FOGSTART                      = 36,
    D3DRS_FOGEND                       = 37,
    D3DRS_FOGDENSITY                   = 38,
    D3DRS_RANGEFOGENABLE               = 48,
    D3DRS_STENCILENABLE                 = 52,
    D3DRS_STENCILFAIL                   = 53,
    D3DRS_STENCILZFAIL                  = 54,
    D3DRS_STENCILPASS                   = 55,
    D3DRS_STENCILFUNC                   = 56,
    D3DRS_STENCILREF                     = 57,
    D3DRS_STENCILMASK                   = 58,
    D3DRS_STENCILWRITEMASK              = 59,
    D3DRS_TEXTUREFACTOR                 = 60,
    D3DRS_WRAP0                         = 128,
    D3DRS_WRAP1                         = 129,
    D3DRS_WRAP2                         = 130,
    D3DRS_WRAP3                         = 131,
```

D3DRS_WRAP4	= 132,
D3DRS_WRAP5	= 133,
D3DRS_WRAP6	= 134,
D3DRS_WRAP7	= 135,
D3DRS_CLIPPING	= 136,
D3DRS_LIGHTING	= 137,
D3DRS_AMBIENT	= 139,
D3DRS_FOGVERTEXMODE	= 140,
D3DRS_COLORVERTEX	= 141,
D3DRS_LOCALVIEWER	= 142,
D3DRS_NORMALIZENORMALS	= 143,
D3DRS_DIFFUSEMATERIALSOURCE	= 145,
D3DRS_SPECULARMATERIALSOURCE	= 146,
D3DRS_AMBIENTMATERIALSOURCE	= 147,
D3DRS_EMISSIVEMATERIALSOURCE	= 148,
D3DRS_VERTEXBLEND	= 151,
D3DRS_CLIPPLANEENABLE	= 152,
D3DRS_POINTSIZE	= 154,
D3DRS_POINTSIZE_MIN	= 155,
D3DRS_POINTSPRITEENABLE	= 156,
D3DRS_POINTSCALEENABLE	= 157,
D3DRS_POINTSCALE_A	= 158,
D3DRS_POINTSCALE_B	= 159,
D3DRS_POINTSCALE_C	= 160,
D3DRS_MULTISAMPLEANTIALIAS	= 161,
D3DRS_MULTISAMPLEMASK	= 162,
D3DRS_PATCHEDGESTYLE	= 163,
D3DRS_DEBUGMONITORTOKEN	= 165,
D3DRS_POINTSIZE_MAX	= 166,
D3DRS_INDEXEDVERTEXBLENDENABLE	= 167,
D3DRS_COLORWRITEENABLE	= 168,
D3DRS_TWEENFACTOR	= 170,
D3DRS_BLENDOP	= 171,
D3DRS_POSITIONDEGREE	= 172,
D3DRS_NORMALDEGREE	= 173,
D3DRS_SCISSORTESTENABLE	= 174,
D3DRS_SLOPESCALEDEPTHBIAS	= 175,
D3DRS_ANTIALIASEDLINEENABLE	= 176,
D3DRS_MINTESSELLATIONLEVEL	= 178,

```

D3DRS_MAXTESSELLATIONLEVEL      = 179,
D3DRS_ADAPTIVETESS_X             = 180,
D3DRS_ADAPTIVETESS_Y             = 181,
D3DRS_ADAPTIVETESS_Z             = 182,
D3DRS_ADAPTIVETESS_W             = 183,
D3DRS_ENABLEADAPTIVETESSELATION = 184,
D3DRS_TWOSIDEDSTENCILMODE       = 185,
D3DRS_CCW_STENCILFAIL            = 186,
D3DRS_CCW_STENCILZFAIL           = 187,
D3DRS_CCW_STENCILPASS           = 188,
D3DRS_CCW_STENCILFUNC            = 189,
D3DRS_COLORWRITEENABLE1         = 190,
D3DRS_COLORWRITEENABLE2         = 191,
D3DRS_COLORWRITEENABLE3         = 192,
D3DRS_BLENDFACTOR               = 193,
D3DRS_SRGBWRITEENABLE           = 194,
D3DRS_DEPTHBIAS                 = 195,
D3DRS_WRAP8                     = 198,
D3DRS_WRAP9                     = 199,
D3DRS_WRAP10                    = 200,
D3DRS_WRAP11                    = 201,
D3DRS_WRAP12                    = 202,
D3DRS_WRAP13                    = 203,
D3DRS_WRAP14                    = 204,
D3DRS_WRAP15                    = 205,
D3DRS_SEPARATEALPHABLENDENABLE  = 206,
D3DRS_SRCBLENDALPHA             = 207,
D3DRS_DESTBLENDALPHA            = 208,
D3DRS_BLENDOPALPHA              = 209,
D3DRS_FORCE_DWORD               = 0x7fffffff
} D3DRENDERSTATETYPE;

```

Параметры:

- ❑ **D3DRS_ZENABLE** — определяет значения буфера глубины. Значение **D3DZB_TRUE** включает Z-буферизацию, а **D3DZB_FALSE** отключает. Флаг **D3DZB_USEW** устанавливает W-буферизацию;
- ❑ **D3DRS_FILLMODE** — значение из перечисляемого типа **D3DFILLMODE**, по умолчанию используется **D3DFILL_SOLID**;

- ❑ `D3DRS_SHADEMODE` — один или несколько элементов `D3DSHADEMODE`. Значение по умолчанию `D3DSHADE_GOURAUD`;
- ❑ `D3DRS_ZWRITEENABLE` — значение `TRUE` позволяет записывать в глубинный буфер, является значением по умолчанию. Если `FALSE` — запись в буфер не производится;
- ❑ `D3DRS_ALPHATESTENABLE` — параметр устанавливается в `TRUE`, чтобы допустить альфа-тестирование пиксела. Значение по умолчанию `FALSE`;
- ❑ `D3DRS_LASTPIXEL` — устанавливает допустимое рисование до последнего пиксела в линии, значение по умолчанию `TRUE`;
- ❑ `D3DRS_SRCBLEND` — член `D3DBLEND`. Значение по умолчанию `D3DBLEND_ONE`;
- ❑ `D3DRS_DESTBLEND` — член `D3DBLEND`. Значение по умолчанию `D3DBLEND_ZERO`;
- ❑ `D3DRS_CULLMODE` — определяет отсечение невидимых граней. Может быть установлен один член из типа `D3DCULL`. Значение по умолчанию `D3DCULL_CCW`;
- ❑ `D3DRS_ZFUNC` — член перечисляемого типа `D3DCMPFUNC`. Значение по умолчанию `D3DCMP_LESSEQUAL`;
- ❑ `D3DRS_ALPHAREF` — определяет альфа-тестирование;
- ❑ `D3DRS_ALPHAFUNC` — член перечисляемого типа `D3DCMPFUNC`. Значение по умолчанию `D3DCMP_ALWAYS`;
- ❑ `D3DRS_DITHERENABLE` — значение `TRUE` служит для разрешения дополнительных установок, по умолчанию `FALSE`;
- ❑ `D3DRS_ALPHABLENDENABLE` — параметр устанавливается в `TRUE`, чтобы допустить альфа-смешивание. Значение по умолчанию `FALSE`. Тип альфа-смешивания определяется в `D3DRS_SRCBLEND` и `D3DRS_DESTBLEND`;
- ❑ `D3DRS_FOGENABLE` — параметр, установленный в значение `TRUE`, допускает работу с туманом. Значение по умолчанию `FALSE`;
- ❑ `D3DRS_SPECULARENABLE` — установка параметра в значение `TRUE` включает зеркальный свет. Значение по умолчанию `FALSE`;
- ❑ `D3DRS_FOGCOLOR` — значение типа `D3DCOLOR`. Значение по умолчанию 0;
- ❑ `D3DRS_FOGTABLEMODE` — формула тумана, которую нужно использовать для пиксельного тумана. Значение по умолчанию `D3DFOG_NONE`;
- ❑ `D3DRS_FOGSTART` — значение, определяющее начальный способ установки тумана. Значением по умолчанию является 0.0f;
- ❑ `D3DRS_FOGEND` — значение, определяющее конечный способ установки тумана. Значением по умолчанию является 1.0f;
- ❑ `D3DRS_FOGDENSITY` — плотность тумана;

- ❑ `D3DRS_RANGEFOGENABLE` — значение `TRUE` разрешает использование тумана. Значением по умолчанию является `FALSE`;
- ❑ `D3DRS_STENCILENABLE` — параметр имеет значение по умолчанию `FALSE`. Значение `TRUE` включает буфер трафарета;
- ❑ `D3DRS_STENCILFAIL` — может быть одной из констант в `D3DSTENCILCAPS`. Значение по умолчанию `D3DSTENCILOP_KEEP`;
- ❑ `D3DRS_STENCILZFFAIL` — может быть одной из констант в `D3DSTENCILCAPS`. Значение по умолчанию `D3DSTENCILOP_KEEP`;
- ❑ `D3DRS_STENCILPASS` — может быть одной из констант в `D3DSTENCILCAPS`. Значение по умолчанию `D3DSTENCILOP_KEEP`;
- ❑ `D3DRS_STENCILFUNC` — может быть одной из констант в `D3DCMPFUNC`. Значение по умолчанию `D3DCMP_ALWAYS`;
- ❑ `D3DRS_STENCILREF` — значение ссылки для теста трафарета. Значение по умолчанию `0`;
- ❑ `D3DRS_STENCILMASK` — маска трафарета. По умолчанию маска имеет значение `0xFFFFFFFF`;
- ❑ `D3DRS_STENCILWRITEMASK` — маска трафарета. По умолчанию маска имеет значение `0xFFFFFFFF`;
- ❑ `D3DRS_TEXTUREFACTOR` — значение является переменной `D3DCOLOR`. Значение по умолчанию — непрозрачный белый (`0xFFFFFFFF`);
- ❑ `D3DRS_WRAP0` — текстурное наложение для набора текстурных координат, значение по умолчанию `0`. Количество текстурных наложений может быть не больше восьми, поэтому существует восемь параметров (`D3DRS_WRAP0—D3DRS_WRAP7`), имеющих по умолчанию нулевые значения;
- ❑ `D3DRS_WRAP1—D3DRS_WRAP7` — текстурные наложения для набора текстурных координат;
- ❑ `D3DRS_CLIPPING` — параметр устанавливается по умолчанию в `TRUE`, чтобы допустить отсечение невидимых сторон примитива. Значение `FALSE` необходимо для отключения отсечения;
- ❑ `D3DRS_LIGHTING` — включает или отключает освещение, значение по умолчанию `TRUE`;
- ❑ `D3DRS_AMBIENT` — цвет окружающего освещения. Эта величина типа `D3DCOLOR`. Значение по умолчанию `0`;
- ❑ `D3DRS_FOGVERTEXMODE` — член `D3DFOGMODE`, определяют туман для вершины. Значение по умолчанию `D3DFOG_NONE`;
- ❑ `D3DRS_COLORVERTEX` — разрешает или запрещает окрашивание вершин, `TRUE` — разрешить, `FALSE` — запретить. Значение по умолчанию `TRUE`;

- ❑ `D3DRS_LOCALVIEWER` — значением по умолчанию является `TRUE`, что решает относительное отражение светового эффекта;
- ❑ `D3DRS_NORMALIZENORMALS` — параметр, установленный в значение `TRUE`, осуществляет автоматическую нормализацию вершины. Значение по умолчанию `FALSE`;
- ❑ `D3DRS_DIFFUSEMATERIALSOURCE` — устанавливает рассеянный свет из значения `D3DMATERIALCOLORSOURCE`. Значение по умолчанию `D3DMCS_COLOR1`;
- ❑ `D3DRS_SPECULARMATERIALSOURCE` — зеркальный источник света. Значение по умолчанию `D3DMCS_COLOR2`;
- ❑ `D3DRS_AMBIENTMATERIALSOURCE` — окружающий источник света. Значение по умолчанию `D3DMCS_MATERIAL`;
- ❑ `D3DRS_EMISSIVEMATERIALSOURCE` — эмиссионный источник света. Значение по умолчанию `D3DMCS_MATERIAL`;
- ❑ `D3DRS_VERTEXBLEND` — параметр имеет значение по умолчанию `D3DVBF_DISABLE`;
- ❑ `D3DRS_CLIPPLANEENABLE` — разрешение или запрещение "клиппинга";
- ❑ `D3DRS_POINTSIZE` — переменная величина, вычисляющая размер точки в случае, когда он не определен для каждой из вершин;
- ❑ `D3DRS_POINTSIZE_MIN` — значение определяет минимальный размер точек в примитиве;
- ❑ `D3DRS_POINTSPRITEENABLE` — булева величина. Значение по умолчанию `FALSE`;
- ❑ `D3DRS_POINTSCALEENABLE` — булева величина. Значение по умолчанию `FALSE`;
- ❑ `D3DRS_POINTSCALE_A` — переменная величина, регулирующая расстояния между точками в примитиве. Значение по умолчанию `1.0f`. Значения всегда больше или равны `0.0f`;
- ❑ `D3DRS_POINTSCALE_B` — переменная величина, регулирующая расстояния между точками в примитиве. Значение по умолчанию `1.0f`. Значения всегда больше или равны `0.0f`;
- ❑ `D3DRS_POINTSCALE_C` — переменная величина, регулирующая расстояния между точками в примитиве. Значение по умолчанию `1.0f`. Значения всегда больше или равны `0.0f`;
- ❑ `D3DRS_MULTISAMPLEANTIALIAS` — булева величина. Значением по умолчанию является `TRUE`;
- ❑ `D3DRS_MULTISAMPLEMASK` — маска. Значение по умолчанию — `0xFFFFFFFF`;

- ❑ `D3DRS_PATCHEDGESTYLE` — член перечисляемого типа `D3DPATCHEDGESTYLE`. Служит для установок стиля тесселяции, значение по умолчанию `D3DPATCHEDGE_DISCRETE`;
- ❑ `D3DRS_DEBUGMONITORTOKEN` — устанавливается для настройки монитора с помощью перечисляемого типа `D3DDEBUGMONITORTOKENS`;
- ❑ `D3DRS_POINTSIZE_MAX` — переменная величина, определяющая максимальный размер. Значение по умолчанию 64.0f;
- ❑ `D3DRS_INDEXEDVERTEXBLENDENABLE` — булева величина, которая разрешает или запрещает индексное смешивание вершин. Значение по умолчанию `FALSE`;
- ❑ `D3DRS_COLORWRITEENABLE` — величина, разрешающая запись цвета в буфер;
- ❑ `D3DRS_TWEENFACTOR` — переменное значение, регулирующее дублирующий фактор. Значение по умолчанию 0.0f;
- ❑ `D3DRS_BLENDOP` — используется при альфа-смешивании для выполнения арифметического действия;
- ❑ `D3DRS_POSITIONDEGREE` — степень интерполяции. Значение по умолчанию `D3DDEGREE_CUBIC`;
- ❑ `D3DRS_NORMALDEGREE` — нормальная степень интерполяции. Значение по умолчанию `D3DDEGREE_LINEAR`;
- ❑ `D3DRS_SCISSORTESTENABLE` — значение `TRUE` разрешает тест на отсечение, значение `FALSE` запрещает. Значение по умолчанию `FALSE`;
- ❑ `D3DRS_SLOPESCALEDPTHBIAS` — определяет смещение;
- ❑ `D3DRS_ANTIALIASEDLINEENABLE` — значение `TRUE` разрешает сглаживание линий, `FALSE` запрещает. Значение по умолчанию `FALSE`;
- ❑ `D3DRS_MINTESSELLATIONLEVEL` — минимальный уровень тесселяции. Значение по умолчанию 1.0f;
- ❑ `D3DRS_MAXTESSELLATIONLEVEL` — максимальный уровень тесселяции. Значение по умолчанию 1.0f;
- ❑ `D3DRS_ADAPTIVETESS_X` — сумма адаптивной тесселяции для координаты *X*. Значением по умолчанию является 0.0f;
- ❑ `D3DRS_ADAPTIVETESS_Y` — сумма адаптивной тесселяции для координаты *Y*. Значением по умолчанию является 0.0f;
- ❑ `D3DRS_ADAPTIVETESS_Z` — сумма адаптивной тесселяции для координаты *Z* направления. Значением по умолчанию является 0.0f;
- ❑ `D3DRS_ADAPTIVETESS_W` — сумма адаптивной тесселяции для координаты *W*. Значением по умолчанию является 0.0f;

- ❑ `D3DRS_ENABLEADAPTIVETESSELATION` — значение `TRUE` позволяет осуществить адаптивную тесселяцию, `FALSE` — запрещает. Значение по умолчанию `FALSE`;
- ❑ `D3DRS_TWOSIDEDSTENCILMODE` — значение `TRUE` разрешает двусторонний способ нанесения по трафарету, `FALSE` — запрещает. Значение по умолчанию `FALSE`;
- ❑ `D3DRS_CCW_STENCILFAIL` — трафаретные операции выполняются, если `CCW` терпит неудачу. Значением по умолчанию является `0x00000001`;
- ❑ `D3DRS_CCW_STENCILZFAIL` — трафаретные операции выполняются, если `CCW` выполняется, а `Z`-тест терпит неудачу. Значением по умолчанию является `0x00000001`;
- ❑ `D3DRS_CCW_STENCILPASS` — трафаретные операции выполняются, если `CCW` и `Z`-тест выполняются. Значением по умолчанию является `0x00000001`;
- ❑ `D3DRS_CCW_STENCILFUNC` — функция сравнения. Значением по умолчанию является `0x00000008`;
- ❑ `D3DRS_COLORWRITEENABLE1` — дополнительное значение `ColorWriteEnable` для устройства. Значением по умолчанию является `0x0000000f`;
- ❑ `D3DRS_COLORWRITEENABLE2` — дополнительное значение `ColorWriteEnable` для устройства. Значением по умолчанию является `0x0000000f`;
- ❑ `D3DRS_COLORWRITEENABLE3` — дополнительное значение `ColorWriteEnable` для устройства. Значением по умолчанию является `0x0000000f`;
- ❑ `D3DRS_BLENDFACTOR` — выставляется когда `D3DCOLOR` используется для константного смешивания при альфа-смешивании. Значением по умолчанию является `0xffffffff`;
- ❑ `D3DRS_SRGBWRITEENABLE` — разрешает рендеринг при гамма-коррекции в формате `sRGB`. Значение по умолчанию `0`;
- ❑ `D3DRS_DEPTHBIAS` — значение с плавающей точкой, для сравнения глубинных величин. Значение по умолчанию `0`;
- ❑ `D3DRS_WRAP8` — второй уровень текстурного наложения для набора текстурных координат;
- ❑ `D3DRS_WRAP9—D3DRS_WRAP15` — вторые уровни текстурного наложения для наборов текстурных координат;
- ❑ `D3DRS_SEPARATEALPHABLENDENABLE` — значение `TRUE` разрешает отдельный способ смешивания для альфа-канала. Значение по умолчанию `FALSE`;
- ❑ `D3DRS_SRCBLENDALPHA` — член перечисляемого типа `D3DBLEND`. Игнорируется если `D3DRS_SEPARATEALPHAENABLE` не верен. Значение по умолчанию `D3DBLEND_ONE`;

- ❑ `D3DRS_DESTBLENDALPHA` — член перечисляемого типа `D3DBLEND`. Игнорируется если `D3DRS_SEPARATEALPHAENABLE` не верен. Значение по умолчанию `D3DBLEND_ZERO`;
- ❑ `D3DRS_BLENDOPALPHA` — значение используется для выбора математических операций при разделении альфа-смешивания. Значение по умолчанию `D3DBLENDOP_ADD`;
- ❑ `D3DRS_FORCE_DWORD` — значение не используется.

D3DVERTEXBLENDFLAGS

Определяет флаги, контролирующие количество матриц. Тип описывается таким образом:

```
typedef enum _D3DVERTEXBLENDFLAGS {
    D3DVBF_DISABLE = 0,
    D3DVBF_1WEIGHTS = 1,
    D3DVBF_2WEIGHTS = 2,
    D3DVBF_3WEIGHTS = 3,
    D3DVBF_TWEENING = 255,
    D3DVBF_0WEIGHTS = 256
} D3DVERTEXBLENDFLAGS;
```

Параметры:

- ❑ `D3DVBF_DISABLE` — отключает смешивание вершин;
- ❑ `D3DVBF_1WEIGHTS` — допускает смешивание вершин двух матриц;
- ❑ `D3DVBF_2WEIGHTS` — допускает смешивание вершин между тремя матрицами;
- ❑ `D3DVBF_3WEIGHTS` — допускает смешивание вершин между четырьмя матрицами;
- ❑ `D3DVBF_TWEENING` — смешивание вершин произойдет с использованием величины, назначенной как `D3DRS_TWEENFACTOR`;
- ❑ `D3DVBF_0WEIGHTS` — использует одну матрицу.

D3DZBUFFERTYPE

Определяет константы, описывающие буфер глубины. Тип описывается таким образом:

```
typedef enum _D3DZBUFFERTYPE {
    D3DZB_FALSE = 0;
    D3DZB_TRUE = 1;
    D3DZB_USW = 2;
```

```
D3DZB_FORSE_DWORD = 0x7fffffff
} D3DZBUFFERTYPE;
```

Параметры:

- ☐ D3DZB_FALSE — не разрешает использовать Z-буфер;
- ☐ D3DZB_TRUE — разрешает использовать Z-буфер;
- ☐ D3DZB_USW — разрешает использовать W-буфер;
- ☐ D3DZB_FORSE_DWORD — значение не используется.

Макросы DirectX Graphics

В DirectX Graphics предопределен ряд макросов для упрощения программирования графики, некоторые из них вы найдете в этом разделе.

D3DCOLOR_ARGB

Инициализирует цвет с альфа-, красной, зеленой, и синей компонентами. Макрос описывается таким образом:

```
#define D3DCOLOR_ARGB(a,r,g,b) \
((D3DCOLOR) (((a)&0xff)<<24) | (((r)&0xff)<<16) | (((g)&0xff)<<8) | \
((b)&0xff)))
```

Параметры:

- ☐ a — альфа-компонент цвета. Величина должна быть в диапазоне от 0 до 255;
- ☐ r — красный компонент цвета. Величина должна быть в диапазоне от 0 до 255;
- ☐ g — зеленый компонент цвета. Величина должна быть в диапазоне от 0 до 255;
- ☐ b — синий компонент цвета. Величина должна быть в диапазоне от 0 до 255.

D3DCOLOR_COLORVALUE

Инициализирует цвет с красной, зеленой, синей и альфа-компонентами. Макрос описывается таким образом:

```
#define D3DCOLOR_COLORVALUE(r,g,b,a) \
D3DCOLOR_RGBA((DWORD)((r)*255.f), (DWORD)((g)*255.f), (DWORD)((b)*255.f), \
(DWORD)((a)*255.f))
```

Параметры:

- ☐ r — красный компонент цвета. Величина должна быть с плавающей точкой в диапазоне от 0.0 до 1.0;

- `g` — зеленый компонент цвета. Величина должна быть с плавающей точкой в диапазоне от 0.0 до 1.0;
- `b` — синий компонент цвета. Величина должна быть с плавающей точкой в диапазоне от 0.0 до 1.0;
- `a` — альфа-компонент цвета. Величина должна быть с плавающей точкой в диапазоне от 0.0 до 1.0.

D3DCOLOR_RGBA

Инициализирует цвет с красной, зеленой, синей и альфа-компонентами. Макрос описывается таким образом:

```
#define D3DCOLOR_RGBA(r,g,b,a) D3DCOLOR_ARGB(a,r,g,b)
```

Параметры:

- `r` — красный компонент цвета. Величина должна быть в диапазоне от 0 до 255;
- `g` — зеленый компонент цвета. Величина должна быть в диапазоне от 0 до 255;
- `b` — синий компонент цвета. Величина должна быть в диапазоне от 0 до 255;
- `a` — альфа-компонент цвета. Величина должна быть в диапазоне от 0 до 255.

D3DTS_WORLDMATRIX

Индексы. Макрос описывается таким образом:

```
#define D3DTS_WORLDMATRIX(index) (D3DTRANSFORMSTATETYPE)(index + 256)
```

Параметр:

- `index` — индексная величина в диапазоне от 0 до 255.

DirectInput

Компонент DirectInput предоставляет возможность работы с устройствами ввода. Рассматриваются необходимые в программировании интерфейсы, функции и структуры.

Интерфейсы DirectInput

В данном списке приведены интерфейсы DirectInput с их кратким описанием:

- `IDirectInput8` — это самый главный интерфейс, и с его создания начинается любая работа с DirectInput. После создания основного интерфейса происходит инициализация всей библиотеки и получение доступа к двум

другим интерфейсам, и как следствие, ко всем структурам, макросам, функциям и т. д.;

- ❑ `IDirectInputDevice8` — это второй по значимости интерфейс, отвечающий за создание устройств ввода: клавиатуры, мыши и джойстика. Под созданием устройства подразумевается настройка, например, клавиатуры, на общее взаимодействие с приложением через интерфейс `IDirectInput8`;
- ❑ `IDirectInputEffect8` — этот интерфейс осуществляет работу с устройствами обратной связи.

Функции `DirectInput`

Раздел содержит информацию об основных функциях для работы с устройствами ввода, компонентами `DirectInput`.

DirectInput8Create()

Создает и инициализирует объект `DirectInput`. Функция описывается таким образом:

```
HRESULT WINAPI DirectInput8Create(
    HINSTANCE hinst,
    DWORD     dwVersion,
    REFIID    riidIls,
    LPVOID*   lpIplDirectInput,
    LPUNKNOWN pUnkOuter);
```

Параметры:

- ❑ `hinst` — дескриптор экземпляра приложения. Этот дескриптор находится в функции `WinMain()`;
- ❑ `dwVersion` — версия `DirectInput`. Это значение обычно `DIRECTINPUT_VERSION`;
- ❑ `riidIls` — версия создаваемого вами интерфейса `IID_DirectInput8`, при желании вы можете обратиться и к более ранней версии;
- ❑ `lpIplDirectInput` — адрес переменной, содержащий созданный указатель на COM-интерфейс;
- ❑ `pUnkOuter` — адрес интерфейса наследования `IUnknown` для использования COM-модели. Обычно равен `NULL`.

IDirectInput8::CreateDevice

Создает и инициализирует устройство. Функция описывается таким образом:

```
HRESULT CreateDevice(
    REFGUID rguid,
```

```
LPDIRECTINPUTDEVICE8* lpIpdirectInputDevice,
LPUNKNOWN              pUnkOuter)
```

Параметры:

- ❑ `rguid` — глобально-уникальный идентификатор GUID создаваемого устройства ввода/вывода. `GUID_SysKeyboard` определен как глобальный идентификатор для клавиатуры;
- ❑ `lpIpdirectInputDevice` — адрес переменной, содержащий указатель на интерфейс `IDirectInputDevice8`;
- ❑ `pUnkOuter` — адрес интерфейса наследования `IUnknown` для использования COM-модели. Обычно равен `NULL`.

IDirectInput8::EnumDevices

Перечисляет доступные устройства. Функция описывается таким образом:

```
HRESULT EnumDevices(
    DWORD                dwDevType,
    LPDIENUMDEVICESCALLBACK lpCallback,
    LPVOID               pvRef,
    DWORD                dwFlags
);
```

Параметры:

- ❑ `dwDevType` — типы устройства;
- ❑ `lpCallback` — адрес функции возврата;
- ❑ `pvRef` — определяет возврат перечислений;
- ❑ `dwFlags` — флаги, определяющие область перечисления.

IDirectInput8::FindDevice

Извлекает глобально-уникальный идентификатор устройства. Функция описывается таким образом:

```
HRESULT FindDevice(
    REFGUID rguidClass,
    LPCTSTR ptszName,
    LPGUID  pguidInstance
);
```

Параметры:

- ❑ `rguidClass` — уникальный идентификатор класса устройства;
- ❑ `ptszName` — название устройства;
- ❑ `pguidInstance` — GUID для устройства, если таковое обнаружено.

IDirectInput8::GetDeviceStatus

Извлекает статус определенного устройства. Функция описывается таким образом:

```
HRESULT GetDeviceStatus(REFGUID rguidInstance);
```

Параметр:

□ `rguidInstance` — глобально-уникальный идентификатор GUID.

IDirectInput8::Initialize

Инициализирует объект `DirectInput`, но вызов этой функции не обязателен, т. к. функция `DirectInput8Create` автоматически инициализирует объект `DirectInput`. Функция описывается таким образом:

```
HRESULT Initialize(
    HINSTANCE hinst,
    DWORD     dwVersion
);
```

Параметры:

□ `hinst` — дескриптор приложения;

□ `dwVersion` — номер версии `DirectInput` для приложения. Всегда используется макрос `DIRECTINPUT_VERSION`.

IDirectInput8::RunControlPanel

Позволяет модифицировать конфигурации. Функция описывается таким образом:

```
HRESULT RunControlPanel(
    HWND hwndOwner,
    DWORD dwFlags
);
```

Параметры:

□ `hwndOwner` — "хендл" родительского окна. Если этот параметр имеет значение `NULL`, то родительское окно не использовано;

□ `dwFlags` — параметр не используется и должен быть установлен в 0.

IDirectInputDevice8::Acquire

Функция описывается таким образом:

```
HRESULT Acquire(VOID)
```

Функция не имеет параметров.

IDirectInputDevice8::BuildActionMap

Формирует карту действий (action map) для устройства и извлекает информацию о нем. Функция описывается таким образом:

```
HRESULT BuildActionMap(  
    LPDIACTIONFORMAT lpdiaf,  
    LPCTSTR          lpszUserName,  
    DWORD            dwFlags  
);
```

Параметры:

- ❑ `lpdiaf` — адрес `DIACTIONFORMAT`, который получает информацию о карте действий;
- ❑ `lpszUserName` — если `NULL`, то используется;
- ❑ `dwFlags` — флаги управления.

IDirectInputDevice8::GetCapabilities

Получает возможности объекта `DirectInputDevice`. Функция описывается таким образом:

```
HRESULT GetCapabilities(LPDIIDEVCAPS lpDIDevCaps);
```

Параметр:

- ❑ `lpDIDevCaps` — адрес структуры `DIDEVCAPS`, которая заполняется данными о возможностях устройства.

IDirectInputDevice8::GetDeviceInfo

Получает информацию об устройстве. Функция описывается таким образом:

```
HRESULT GetDeviceInfo(LPDIDEVICEINSTANCE pdidi);
```

Параметр:

- ❑ `pdidi` — адрес структуры `DIDeviceInstance`, которая заполняется информацией об устройстве.

IDirectInputDevice8::GetDeviceState

Получение данных от устройства. Функция описывается таким образом:

```
HRESULT GetDeviceState(  
    DWORD  cbData,  
    LPVOID lpvData);
```

Параметры:

- ❑ `cbData` — размер массива, в который помещаются данные, получаемые от устройства;
- ❑ `lpvData` — адрес массива с данными.

IDirectInputDevice8::SetDataFormat

Устанавливает формат данных. Функция описывается таким образом:

```
HRESULT SetDataFormat(LPCDIDATAFORMAT lpdf);
```

Параметр:

- ❑ `lpdf` — указатель на структуру формата данных, позволяющий задавать способы форматирования устройства клавиатуры, мыши, джойстика. Можно воспользоваться флагами для определения формата данных по умолчанию:
 - `c_dfDIKeyboard` — клавиатура;
 - `c_dfDIMouse` — мышь;
 - `c_dfDIMouse2` — вторая мышь;
 - `c_dfDIJoystick` — джойстик;
 - `c_dfDIJoystick2` — второй джойстик.

IDirectInputDevice8::SetCooperativeLevel

Устанавливает уровень взаимодействия устройства. Функция описывается таким образом:

```
HRESULT SetCooperativeLevel(
    HWND hwnd,
    DWORD dwFlags)
```

Параметры:

- ❑ `hwnd` — дескриптор окна приложения, находящийся в функции `WinMain()`;
- ❑ `dwFlags` — параметр, имеет флаги, определяющие уровень взаимодействия с устройством. Перечислим их:
 - `DISCL_BACKGROUND` — получает второстепенный доступ;
 - `DISCL_EXCLUSIVE` — получает эксклюзивный доступ. Никакое другое приложение не сможет получить доступ к устройству;
 - `DISCL_FOREGROVND` — получает первоочередной доступ;
 - `DISCL_NONEXCLUSIVE` — получает неисключительный доступ;
 - `DISCL_NOWINKEY` — отключает клавишу <ПУСК> в Windows. В исключительном режиме <ПУСК> тоже автоматически отключается.

Структуры DirectInput

В этом разделе приведены структуры интерфейса DirectInput, использованные на уроках 15 и 16.

DIDATAFORMAT

Описывает формат данных устройства. Структура описывается таким образом:

```
typedef struct DIDATAFORMAT {  
    DWORD          dwSize;  
    DWORD          dwObjSize;  
    DWORD          dwFlags;  
    DWORD          dwDataSize;  
    DWORD          dwNumObjs;  
    LPDIOBJECTDATAFORMAT rgodf;  
} DIDATAFORMAT, *LPDIDATAFORMAT;  
  
typedef const DIDATAFORMAT *LPCDIDATAFORMAT;
```

Параметры:

- `dwSize` — размер структуры в байтах;
- `dwObjSize` — размер структуры `DIOBJECTDATAFORMAT` в байтах;
- `dwFlags` — флаги, описывающие атрибуты данных;
- `dwDataSize` — размер пакета данных, возвращаемых устройством, в байтах;
- `dwNumObjs` — количество объектов в массиве структуры `DIOBJECTDATAFORMAT`, адрес которой содержится в параметре `rgodf`;
- `rgodf` — адрес массива структуры `DIOBJECTDATAFORMAT`.

DIMOUSESTATE

Описывает состояние устройства мыши. Структура описывается таким образом:

```
typedef struct _DIMOUSESTATE  
{  
    LONG  lX;  
    LONG  lY;  
    LONG  lZ;  
    BYTE  rgbButtons[4];  
} DIMOUSE *LPDIMOUSESTATE;
```

Параметры:

- `lX` — значение по оси X ;
- `lY` — значение по оси Y ;

- `lz` — значение по оси *Z*;
- `rgbButtons[4]` — с помощью этого параметра происходит обработка нажатия кнопок мыши.

DirectMusic

Компонент **DirectMusic** позволяет работать со звуковыми файлами, упрощая процесс воспроизведения множества звуковых форматов.

Интерфейсы DirectMusic

В данном списке приведены интерфейсы **DirectMusic** с их кратким описанием:

- **IDirectMusic8** — самый главный интерфейс, но его создание не обязательно в силу реализации COM-модели;
- **IDirectMusicPerformance8** — интерфейс, с помощью которого происходит управление музыкальными данными в создаваемом вами приложении. Также он создает объект **IDirectMusic** и это, опять-таки, связано с COM-моделью;
- **IDirectMusicLoader8** — это интерфейс, необходимый для создания загрузчика, при помощи которого вы подгрузите музыкальный файл в ваше приложение;
- **IDirectMusicSegment8** — все музыкальные данные загружаются в раздельно-сегменты, этот интерфейс отвечает за правильное представление сегментных данных.

Функции DirectMusic

Большое количество функций интерфейса **DirectMusic**, в основном предназначенных для загрузки и воспроизведения звуковых данных. Некоторые из них находятся в этом разделе.

IDirectMusicLoader8::LoadObjectFromFile

Загружает файл. Функция описывается таким образом:

```
HRESULT LoadObjectFromFile(  
    REFGUID rguidClassID,  
    REFIID iidInterfaceID,  
    WCHAR* pwzFilePath,  
    void** ppObject  
);
```

Параметры:

- ❑ `rguidClassID` — в этом параметре необходимо указать уникальный идентификатор для объекта, отвечающего за сегмент. Используется идентификатор класса `CLSID_DirectMusicSegment`;
- ❑ `iidInterfaceID` — здесь необходимо указать уникальный идентификатор для интерфейса сегмента. Для загрузки файла в сегмент используется `IID_IDirectMusicSegment8`;
- ❑ `pwzFilePath` — название музыкального файла для воспроизведения;
- ❑ `ppObject` — адрес указателя на интерфейс `IDirectMusicSegment8`.

IDirectMusicPerformance8::InitAudio

Инициализирует аудиосистему. Функция описывается таким образом:

```
HRESULT InitAudio(
    IDirectMusic**      ppDirectMusic,
    IDirectSound**      ppDirectSound,
    HWND                hWnd,
    DWORD                dwDefaultPathType,
    DWORD                dwPChannelCount,
    DWORD                dwFlags,
    DMUS_AUDIOPARAMS*   pParams
);
```

Параметры:

- ❑ `ppDirectMusic` — адрес переменной, получающей указатель интерфейса на объект `DirectMusic`;
- ❑ `ppDirectSound` — здесь та же самая ситуация, что и в первом параметре, за исключением того, что все действия связаны с `DirectSound`;
- ❑ `hWnd` — это "хендл" окна;
- ❑ `dwDefaultPathType` — эта переменная определяет аудиопуть;
- ❑ `dwPChannelCount` — в этой переменной указывается число каналов;
- ❑ `dwFlags` — флаги, с помощью которых устанавливаются эффекты звучания;
- ❑ `pParams` — адрес структуры `DMUS_AUDIOPARAMS`, задающей параметры синтезатору. Можно установить значение `NULL`, и эта структура не будет востребована.

IDirectMusicPerformance8::PlaySegmentEx

Воспроизводит сегмент. Функция описывается таким образом:

```
HRESULT PlaySegmentEx(
    IUnknown*            pSource,
```

```

WCHAR*                pwzSegmentName,
IUnknown*              pTransition,
DWORD                 dwFlags,
__int64               i64StartTime,
IDirectMusicSegmentState** ppSegmentState,
IUnknown*             pFrom,
IUnknown*             pAudioPath
);

```

Параметры:

- ❑ `pSource` — адрес указателя на сегмент `pSegment`;
- ❑ `pwzSegmentName` — этот параметр в DirectX 9 не используется;
- ❑ `pTransition` — этот параметр позволяет настраивать переход к сегменту. Может быть установлен в `NULL`;
- ❑ `dwFlags` — очень большое количество флагов из перечисляемого типа `DMUS_SEGF_FLAGS`, позволяющих настраивать воспроизведение файла. Но можно установить нулевое значение, чтобы игнорировать этот параметр;
- ❑ `i64StartTime` — время воспроизведения сегмента. Если поставить 0, воспроизведение произойдет моментально;
- ❑ `ppSegmentState` — позволяет настраивать состояние сегмента, обычно используется значение `NULL`;
- ❑ `pFrom` — с помощью этого параметра можно остановить воспроизведение сегмента в момент начала проигрыша другого сегмента. Можно поставить значение `NULL`;
- ❑ `pAudioPath` — указатель на интерфейс `IUnknown`, определяет аудиопуть.

IDirectMusicSegment8::SetRepeats

Повтор воспроизведения. Функция описывается таким образом:

```
HRESULT SetRepeats(DWORD dwRepeats)
```

- ❑ `dwRepeats` — единственный параметр, отвечающий за воспроизведение файла. Можно установить цифры 1, 2, 3 и т. д. Или воспользоваться флагом `DMUS_SEG_REPEAT_INFINITE` для бесконечного повтора. Цифра 0 указывает на один-единственный проигрыш.

DirectSound

Благодаря компоненту DirectSound существует возможность создания основных, вспомогательных, статических и потоковых буферов для обеспечения всевозможных эффектов звучания.

Интерфейсы DirectSound

В данном списке приведены интерфейсы DirectSound с их кратким описанием:

- `IDirectSound8` — создает и настраивает буфер;
- `IDirectSoundBuffer8` — управляет буфером;
- `IDirectSound3DBuffer8` — устанавливает параметры и описывает ориентацию 3D-буфера.

Функции DirectSound

Набор функций DirectSound позволяет создавать и работать с буферами устройства, загружая при этом необходимые звуковые данные.

DirectSoundCreate8()

Создает главный объект DirectSound. Функция описывается таким образом:

```
HRESULT DirectSoundCreate8(  
    LPGUID          lpGuidDevice  
    LPDIRECTSOUND*  ppDS8  
    IUnknown FAR    pUnkOuter)
```

Параметры:

- `lpGuidDevice` — содержит GUID звуковой карты. В том случае, когда в системе присутствует одна карта, можно установить `NULL` в качестве значения по умолчанию;
- `ppDS8` — указатель на главный объект DirectSound;
- `pUnkOuter` — обычно его значение равно `NULL`.

IDirectSound8::CreateSoundBuffer

Создает вторичный буфер. Функция описывается таким образом:

```
HRESULT CreateSoundBuffer(  
    LPCDSBUFFERDESC  pcDSBufferDesc,  
    LPDIRECTSOUNDBUFFER ppDSBuff,  
    IUnknown FAR*    pUnkOuter)
```

Параметры:

- `pcDSBufferDesc` — структура, описывающая создаваемый буфер. Прежде чем вызвать эту функцию, вы должны описать структуру `DSBUFFERDESC`;
- `ppDSBuff` — указатель на `IDirectSoundBuffer8`;

- `pUnkOuter` — содержит указатель на интерфейс `IUnknown` объекта-агрегата. Поскольку агрегация не используется, всегда устанавливается значение `NULL`.

IDirectSound8::SetCooperativeLevel

Устанавливает уровень взаимодействия. Функция описывается таким образом:

```
HRESULT SetCooperativeLevel(  
    HWND hwnd,  
    DWORD dwLevel)
```

Параметры:

- `hwnd` — дескриптор окна приложения;
- `dwLevel` — флаги, устанавливающие уровень взаимодействия. Перечислим их:
 - `DSSCL_NORMAL` — стандартный уровень кооперации;
 - `DSSCL_PRIORITY` — приоритетный уровень кооперации;
 - `DSSCL_EXCLUSIVE` — эксклюзивный доступ;
 - `DSSCL_WRITEPRIMARY` — полный контроль.

IDirectSoundBuffer8::Lock

Блокирует буфер. Функция описывается таким образом:

```
HRESULT Lock(  
    DWORD dwOffset,  
    DWORD dwBytes,  
    LPVOID* ppvAudioPtr1,  
    LPDWORD pdwAudioBytes1,  
    LPVOID* ppvAudioPtr2,  
    LPDWORD pdwAudioBytes2,  
    DWORD dwFlags  
);
```

Параметры:

- `dwOffset` — расположение курсора записи;
- `dwBytes` — размер блокируемого буфера в байтах;
- `ppvAudioPtr1` — адрес указателя на массив первой части данных;
- `pdwAudioBytes1` — адрес размера массива первой части данных;
- `ppvAudioPtr2` — адрес указателя на массив второй части данных;

- `pdwAudioBytes2` — адрес размера массива второй части данных;
- `dwFlags` — флаги, определяющие способ блокировки буфера. Имеется только два флага:
 - `DSBLOCK_ENTIREBUFFER` — блокируется весь буфер;
 - `DSBLOCK_FROMWRITECURSOR` — блокируется только часть буфера исходя из позиции курсора записи.

IDirectSoundBuffer8::Unlock

Разблокирует буфер. Функция описывается таким образом:

```
HRESULT Unlock(  
    LPVOID pvAudioPtr1,  
    DWORD dwAudioBytes1,  
    LPVOID pvAudioPtr2,  
    DWORD dwAudioBytes2  
);
```

Параметры:

- `pvAudioPtr1` — адрес указателя на первую часть данных буфера;
- `dwAudioBytes1` — размер первой части данных в байтах;
- `pvAudioPtr2` — адрес указателя на вторую часть данных буфера;
- `dwAudioBytes2` — размер второй части данных буфера в байтах.

IDirectSoundBuffer8::Play

Воспроизводит данные из буфера. Функция описывается таким образом:

```
HRESULT Play(  
    DWORD dwReserved1,  
    DWORD dwPriority,  
    DWORD dwFlags)
```

Параметры:

- `dwReserved1` — зарезервированный параметр, должен иметь нулевое значение;
- `dwPriority` — установка приоритета воспроизведения, всегда используется значение `NULL`;
- `dwFlags` — параметр, определяющий схему воспроизведения. Имеет флаг `DSBPLAY_LOOPING`, зацикливающий воспроизведение. Можно установить целочисленное значение, указывающее число повторений воспроизведения. Ноль соответствует одному разу.

Структуры DirectSound

В DirectSound имеются различные виды структур. Рассмотрим две структуры, а также разберем их параметры.

DSBUFFERDESC

Описывает характеристики буфера. Структура описывается таким образом:

```
typedef struct _DSBUFFERDESC {
    DWORD          dwSize;
    DWORD          dwFlags;
    DWORD          dwBufferBytes;
    DWORD          dwReserved;
    LPWAVEFORMATEX lpwfxFormat;
    GUID           guid3DAlgorithm;
} DSBUFFERDESC, *LPDSBUFFERDESC;
```

Параметры:

- ☐ `dwSize` — размер создаваемой структуры;
- ☐ `dwFlags` — флаги, с помощью которых настраивается создаваемый буфер. Рассмотрим некоторые из них:
 - `DSBCAPS_CTRLDEFAULT` — настройки по умолчанию;
 - `DSBCAPS_STATIC` — буфер будет содержать статические данные;
 - `DSBCAPS_LOCSOFTWARE` — буфер будет размещаться в программной памяти;
 - `DSBCAPS_PRIMARYBUFFER` — этот флаг создаст один первичный буфер, и вы обязаны настроить этот буфер "вручную".
- ☐ `dwBufferBytes` — размер создаваемого буфера в байтах;
- ☐ `dwReserved` — зарезервированный параметр, должен быть `NULL`;
- ☐ `lpwfxFormat` — с помощью этого параметра описывается формат звуковых данных, содержащихся в создаваемом аудиобуфере. Формат задается с помощью структуры `WAVEFORMATEX`. Заполнение полей этой структуры должно произойти до описания структуры `DSBUFFERDESC`;
- ☐ `guid3DAlgorithm` — уникальный идентификатор алгоритма для использования аппаратной эмуляции, значение по умолчанию `DS3DALG_DEFAULT`.

WAVEFORMATEX

Описывает формат звука. Структура описывается таким образом:

```
typedef struct _WAVEFORMATEX{
    WORD    wFormatTag;
```

```
WORD  nChannels;  
DWORD nSamplesPerSec;  
DWORD nAvgBytesPerSec;  
WORD  nBlockAlign;  
WORD  wBitsPerSample;  
WORD  cbSize;  
} WAVEFORMATEX;
```

Параметры:

- ❑ `wFormatTag` — всегда используется флаг `WAVE_FORMAT_PCM`;
- ❑ `nChannels` — количество создаваемых каналов;
- ❑ `nSamplesPerSec` — частота оцифровки звука;
- ❑ `nAvgBytesPerSec` — скорость воспроизводимых данных;
- ❑ `nBlockAlign` — блочное выравнивание в байтах;
- ❑ `wBitsPerSample` — число бит в сэмпле;
- ❑ `cbSize` — дополнительная информация, но чаще всего этот параметр не используется.

Приложение 2



Web-ресурсы

Из русскоязычных ресурсов, посвященных библиотеке DirectX и программированию компьютерных игр, пожалуй, стоит выделить сайт <http://www.gamedev.ru>. Этот большой и известный сайт существует в сети достаточно продолжительное время и поддерживается большим количеством программистов, завязанных на разработке компьютерных игр. Сайт содержит огромное количество документации и статей на различные темы. Там же можно найти много исходных кодов для DirectX и Open GL. Но самое главное — это потрясающий форум. Без стеснения можно задать вопрос любой сложности, и вы обязательно получите массу ответов и предложений по способу решения вашей проблемы. И тот факт, что в рубрике "Работа" этого сайта постоянно размещаются объявления об имеющихся вакансиях крупных компаний, говорит сам за себя.

Еще хочу привлечь ваше внимание к сайту <http://www.firststeps.ru>. Такого количества информации по программированию в целом, да еще находящейся в одном месте, я не встречал нигде. Сайт также поддерживается несколькими программистами, и вы даже можете приобрести всю версию сайта на компакт-диске.

Из англоязычных ресурсов стоит отметить двух "монстров": <http://www.gamedev.net> и <http://www.gamasutra.com>.

А также сообщаю вам адрес своей электронной почты: DirectX9@mail.ru, на случай, если вдруг вы не сможете найти ответ на имеющийся у вас вопрос. Предполагаю, что количество вопросов будет огромным. На все ответить я не в силах, но на действительно тупиковые постараюсь обязательно дать ответ.

Приложение 3



Список использованных источников

1. Онлайн-овая документация DirectX 9 и Windows 32 API с сайта корпорации Microsoft: <http://www.microsoft.com>.
2. Документация DirectX 9 SDK.

Приложение 4



Описание компакт-диска

Прилагаемый к книге компакт-диск в папках Urok1...Urok19 содержит все примеры и исходные коды из книги.

К большому нашему сожалению на этом компакт-диске Вы не найдете установочного файла пакета DirectX 9 SDK. Почти двухмесячная переписка с корпорацией Microsoft по поводу включения в компакт-диск пакета DirectX 9 SDK, не увенчалась успехом. Мы пытались убедить корпорацию Microsoft, всеми силами объясняя, что в наших (российских) условиях не всем по силам скачать файл размером в 230 Мбайт.

Но, к сожалению, наши призывы и мольбы о помощи не были услышаны, поэтому Вам придется самостоятельно скачивать дистрибутив DirectX 9 SDK с сайта корпорации Microsoft. Чуть ниже этого текста Вы найдете ссылку на сайт корпорации Microsoft, откуда Вам придется осуществить долгий и трудоемкий процесс по скачиванию DirectX 9 SDK.

Мы приносим читателям извинения за доставленные неудобства, но обстоятельства выше нас.

С уважением, автор и издательство "БХВ-Петербург"

<http://www.microsoft.com/downloads/details.aspx?FamilyID=fd044a42-991>

Предметный указатель

D

D3DDEVTYPE 46
DirectInput 233
DirectMusic 263
DirectSound 275
DirectX 21

G, H, Z

GUID 237
HAL 23
Z-буфер 115

A

Альфа-блендинг 191

Б

Буфер:
 вторичный 275
 первичный 275

В

Вершина 27
Вывод текста на экран 152

Д

Данные:
 буферизированные 239
 непосредственные 233
Двойная буферизация 40
Директива:
 pragma 37
Добавление пустого файла 6

И

Инициализация Direct3D 9 36
Интерфейс:
 DirectDraw 25
 IDirect3D9 28
 IDirect3DDevice9 28
 IDirect3DIndexBuffer9 28
 IDirect3DTexture9 29
 IDirect3DVertexBuffer9 28
 IDirect3DVertexShader9 29
 IUnknown 24
 создание указателя 29
Источник света:
 направленный 133
 прожекторный 133
 точечный 133

К

Класс 296
 CSound 284
 CSoundManager 286
 CStreamingSound 287
 CWaveFile 288
 Windowsclass 9

Компиляция 20

Конвейер:

графический 212

программируемый 214

рендеринга 26

функционально-фиксированный 213

М

Материал 131

Матрица 77

вида 82

вращения 81

масштабирования 82

мировая 80

проекции 83

трансляции 81

Микшер 276

Мультитекстурирование 189

Мэш 196

О

Обработка:

ошибок 47

событий 15

Освещение 26

П

Подключение библиотеки в проект 36

Полигон 26

Полноэкранный режим 158

Примитив 67

Р

Растеризация 26

Регистр:

адресный 217

временной 217

входной 219

выходной 219

константный 217

Рендеринг 26

С

Свет 133

Система координат 57

текстурная 172

Создание:

буфера вершин 62

индексного буфера 103

окна 13

проекта 5

с помощью мастера 293

COM 23

Структура:

CUSTOMVERTEX 59

D3DCOLORVALUE 132

D3DDISPLAYMODE 40

D3DLIGHT9 133

D3DLIGHTTYPE 134

D3DMATERIAL9 131

D3DMATRIX 84

D3DPRESENT_PARAMETERS 43

D3DRECT 48

D3DVERTEXELEMENT9 221

D3DXMATRIXA16 86

D3DXVECTOR3 92

DIMOUSESTATE 255

DSBUFFERDESC 279

RECT 152

WAVEFORMATEX 280

WINDCLASSEX 9

Сцена 26, 49

построение 27

Т

Текстурирование 171

Тесселяция 26

Тип:

D3DCULL 119

D3DFORMAT 40

D3DPOOL 63

D3DRENDERSTATETYPE 116

D3DTEXTUREOP 176

D3DTEXTURESTAGESTATETYPE 176

D3DZBUFFERTYPE 118

Трансформация 26

Ф

Формат:

MIDI 263

WAV 263

вершины 59

Функция:

BeginScene 49

CoCreateInstance 265

CoInitialize 265

CreateFont 154

CreateInput 259

CreateKeyboard 259

CreateMouse 261

CreatWindowEx 13

D3DXAssembleShaderFromFile 223

D3DXCreateFont 155

D3DXCreateTextureFromFile 173

D3DXCreateTextureFromFileEx 192

D3DXLoadMeshFromX 198

D3DXMatrixLookAtLH 91

D3DXMatrixMultiply 82, 122

D3DXMatrixRotationX 91

D3DXMatrixScale 82

D3DXMatrixTranslation 81

D3DXVec3Normalize 140

D3DXMatrixPerspectiveFovLH 93

DeleteDirect3D 52

DeleteKeyboard 261

DeleteMouse 261

Direct3DCreate9 39

DirectInput8Create 235

DirectSoundCreate8 277

EndScene 50

GetMessage 16

GetSystemMetrics 158

ID3DXBaseMesh:

DrawSubset 201

ID3DXFont:

DrawText 156

IDirect3D9:

GetAdapterDisplayMode 42

IDirect3DDevice9:

Clear 48

Present 50

CreateVertexBuffer 62

SetStreamSource 66

SetFVF 66

DrawPrimitive 67

SetRenderState 88

SetTransform 91

SetIndices 106

DrawIndexPrimitive 106

SetMaterial 139

SetLight 140

LightEnable 141

SetTexture 175

SetTextureStageState 175

GetDeviceCaps 222

CreateVertexDeclaration 223

CreateVertexShader 223

SetVertexShaderConstantF 224

SetVertexShader 224

SetVertexDeclaration 225

IDirect3DDevice9:

CreateDevice 45

IDirect3DIndexBuffer9:

CreateIndexBuffer 105

IDirect3DVertexBuffer:

Lock 64

Unlock 64

IDirectInput8:

CreateDevice 242

IDirectInputDevice8:

SetDataFormat 243

SetCooperativeLevel 244

Acquire 245

GetDeviceState 245

Unacquire 251

IDirectMusicLoader8:

SetSearchDirectory 268

LoadObjectFromFile 269

IDirectMusicPerformance8:

InitAudio 267

PlaySegmentEx 270

IDirectMusicSegment8:

Download 270

SetRepeats 271

IDirectSound8:

SetCooperativeLevel 277

CreateSoundBuffer 278

IDirectSoundBuffer8:

Lock 281

Unlock 283

Play 283

Stop 283

(окончание рубрики см. на с. 384)

Функция (окончание):

InitialBufferVershin 61
InitialInput 247
InitialMesh 197
InitialObject 104
IntialDirect3D 39
LoadIcon 11
MainWinProc 15
Matrix 90
PeekMessage 68
QueryInterface 25
RegisterClassEx 12
RenderingDirect3D 48
timeGetTime 90
WinMain 9
ZeroMemory 43

Х

Х-файл 195

Ц

Цветовой ключ 191

Ш

Шейдер 215
 вершинный 216
 декларация 221
 пиксельный 227