Алексей Попов

bhv

DirectX 10 — это просто

Программируем графику на C++

Использование Direct3D 10 для вывода графики

Программирование вершинных, пиксельных и геометрических шейдеров на языке HLSL

Быстрое создание приложений с помощью библиотеки DXUT (DirectX Utility)

Справочник функций DirectX 10 и библиотеки DXUT

+cd

Алексей Попов

Программируем графику на C++

Санкт-Петербург

«БХВ-Петербург»

2008

УДК 681.3.06
ББК 32.973.26-018.1
П58

Попов А. А.

П58 DirectX 10 — это просто. Программируем графику на C++. — СПб.: БХВ-Петербург, 2008. — 464 с.: ил. + CD-ROM — (Внесерийная)

ISBN 978-5-9775-0139-2

Рассмотрено создание графических приложений для Windows на C++ с использованием новой версии компонента Direct3D 10, входящего в состав библиотеки DirectX 10. Описывается вывод двухмерной и трехмерной графики, связанное с этим программирование вершинных, пиксельных и геометрических шейдеров на языке HLSL. Отдельная часть книги посвящена быстрой разработке приложений с помощью библиотеки DXUT (DirectX Utility). Материал сопровождается практическими примерами, а также справочником функций библиотек DirectX 10 и DXUT. На прилагаемом компакт-диске находятся все исходные тексты примеров, рассмотренных в книге.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Юрий Якубович*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн обложки	*Инны Тачиной*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.11.07.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 37,41.
Тираж 2500 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой по надзору в сфере защиты прав потребителей и благополучия человека.

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

ISBN 978-5-9775-0139-2

Оглавление

Введение **1**

Кому адресована эта книга? 1
Как пользоваться этой книгой? 1
Что нам понадобится? 3
Благодарности 3

ЧАСТЬ I. ПОСТРОЕНИЕ ПРИЛОЖЕНИЙ НА ОСНОВЕ DIRECTX 10 ..... **5**

Глава 1. Собираемся в путь **7**

Что такое DirectX и зачем он нужен? 7
Немного истории: от Windows 95 к Windows Vista 8
Сравнение DirectX 9 и DirectX 10 10
Где достать DirectX 10 SDK? 11
Установка DirectX 10 SDK на компьютер 11
Среда разработки Visual Studio 2005 14

Глава 2. Первые шаги **21**

Многозадачная операционная система Windows 21
Минимальное приложение Windows 23
Минимальное приложение Direct3D 10 41

Глава 3. Немного математики **71**

Системы координат в DirectX 10 71
Геометрические преобразования 74
Векторы 76
Матрицы 83
Матрицы геометрических преобразований 88
Матрицы в Direct3D 10 90

Глава 4. Двухмерная графика 97

Вывод текста 97
Ресурсы в DirectX 10 113
Выводим картинку 115

Глава 5. Слово о шейдерах 139

Что такое шейдеры? 139
Язык HLSL 142

Глава 6. Запускаем Direct3D 10 157

Рисуем треугольник 157
Анимируем треугольник 184
Больше цвета! 197

Глава 7. Больше реализма: освещение, материалы, текстуры 205

Как включить свет? 205
Модель освещения 209
Полигон для испытаний 212
Направленный источник света 228
Точечный источник света 232
Прожектор 238
Материалы 247
Текстуры 247

Глава 8. Геометрические шейдеры 259

Особенности геометрического шейдера 259
Пишем геометрический шейдер 263
Шаблонный буфер глубины 267

ЧАСТЬ II. ИСПОЛЬЗОВАНИЕ DXUT 10 277

Глава 9. Введение в DXUT 279

Что такое DXUT 279
Первый проект с DXUT 279
Вывод текста на экран 289

Глава 10. Загружаем объемные модели 299

Как создать файл формата .sdkmesh? 300
Вывод модели на экран 303
Камера для просмотра модели 320

Глава 11. Пользовательский интерфейс 327

Обзор элементов интерфейса 327
Добавляем интерфейс в программу 330

Глава 12. Диалоговые окна 369

Создание диалогового окна 369
Окно настройки параметров устройства Direct3D 374

ПРИЛОЖЕНИЯ 379

Приложение 1. Краткий справочник по DirectX10 381

Интерфейсы 381
Функции 401
Структуры 411

Приложение 2. Справочник функций DXUT 423

Инициализация DXUT 423
Установка функций обратного вызова 427
Выполнение стандартных задач 431
Получение сведений об установках DXUT 437
Структуры 447

Приложение 3. Описание компакт-диска 452

Предметный указатель 453

Введение

Будущее надвигается стремительно: те компьютеры, которые несколько лет назад были пределом мечтаний, сегодня вызывают лишь усмешку. Быстродействие процессоров все растет, увеличивается объем оперативной памяти и емкость жестких дисков. Технологии тоже развиваются, стремятся использовать возможности нового "железа" на все 100%. За период времени, чуть больший двух десятилетий, был пройден путь от перфокарт до виртуальной реальности. Что ждет нас впереди? К чему готовиться? В этой книге вы можете познакомиться с библиотекой DirectX 10, которая, очевидно, получит широкое распространение в ближайшем будущем. Эта книга о трехмерной графике. Если лично вам было бы интересно уже сегодня прикоснуться к графической библиотеке, на которой, скорее всего, будут построены еще не известные игровые хиты будущего, то вы взяли с полки правильную книгу.

Кому адресована эта книга?

Книга ориентирована на начинающего программиста, имеющего опыт написания программ на языке C++ хотя бы для MS-DOS. Желательно знакомство со средой разработки Visual Studio, хотя краткое описание работы с ней приводится в книге. Дополнительный опыт составления программ для Windows, а также работы с предыдущими версиями DirectX пойдет только на пользу.

Как пользоваться этой книгой?

В книге содержится как учебный (*части I* и *II*), так и справочный материал (*приложения*). Вопросы рассматриваются по мере увеличения их сложности, поэтому желательно знакомиться с главами в том порядке, в котором они расположены в книге. В *части I* мы рассматриваем общие принципы

программирования под Windows и создания приложений Direct3D 10. Вот в каком порядке это излагается.

- *Глава 1* содержит вводные сведения о DirectX, а также расскажет о том, как установить на компьютер DirectX SDK и настроить среду разработки для компиляции программ.
- *Глава 2* познакомит с принципом работы приложений Windows и научит создавать минимальное приложение Direct3D 10.
- *Глава 3* введет в курс некоторых элементов математики, которые используются при программировании трехмерной графики.
- *Глава 4* научит выводить на экран текст и изображения из файла.
- *Глава 5* подготовит к программированию шейдеров с использованием языка HLSL.
- *Глава 6* расскажет, как вывести на экран треугольник и анимировать его.
- *Глава 7* покажет, как добавить в программу расчет освещения и наложить на поверхности текстуры.
- *Глава 8* познакомит с нововведением десятой версии DirectX с геометрическими шейдерами.

В *части II* рассматривается создание программ с использованием каркаса DXUT, который позволяет отвлечься от таких рутинных операций, как создание окон или устройств Direct3D, и сосредоточиться непосредственно на реализации алгоритма программы. Вот какие темы освещаются в главах второй части.

- *Глава 9* содержит общие сведения о программном каркасе DXUT и его функционировании.
- *Глава 10* научит, как загружать и выводить на экран трехмерные модели средствами DXUT.
- *Глава 11* расскажет о работе с интерфейсом пользователя, который DXUT предоставляет в распоряжение программиста.
- *Глава 12* познакомит с созданием собственных диалоговых окон DXUT, а также с использованием стандартного окна настроек Direct3D.

В *приложении 1* находится справочник по функциям DirectX 10, в *приложении 2* можно познакомиться с некоторыми полезными функциями DXUT. Таким образом, книгу можно использовать и как учебник, и как справочник для разбора примеров программ. Также нужно отметить, что в книге используется стиль, который можно условно назвать "стиль университетских лекций", поэтому, если вы видите слово "вы", написанное со строчной буквы, не принимайте это как знак неуважения, это всего лишь обращение к аудитории.

Что нам понадобится?

Поскольку библиотека DirectX 10 в настоящее время имеется только для Windows Vista, для работы нам понадобится компьютер с этой операционной системой. Чтобы в полной мере оценить возможности новой версии DirectX, лучше также иметь видеокарту, аппаратно поддерживающую DirectX 10, однако обязательным условием это не является.

Все программы, описанные в книге, были составлены и скомпилированы в среде Visual Studio 2005. Предыдущая версия, Visual Studio 2003, тоже должна работать, однако это не проверялось. Программы компилировались с использованием DirectX SDK за август 2007 года.

Благодарности

Хочу выразить огромную признательность моей любимой жене Наташе за то, что она поддерживала меня во время подготовки книги и создавала все условия для того, чтобы она увидела свет.

Своих родителей я благодарю за то, что с детства прививали мне интерес ко всему новому и учили мыслить самостоятельно.

Слова благодарности я также должен сказать моим школьным учителям, в особенности Ивашевой Светлане Валерьевне, учительнице математики, и Смолко Светлане Георгиевне, учительнице русского языка и литературы. Они научили меня, как я надеюсь, излагать свои мысли на родном языке. Если вдруг в книге что-то изложено не логично, или просто "не по-русски", то это только потому, что я забыл, чему они меня учили. Еще обязательно нужно поблагодарить Белоусову Любовь Юрьевну, учительницу по информатике, за то, что она поддержала мой интерес к компьютерной технике, когда я учился в школе: не будь ее понимания и поддержки в то время, эта книга, скорее всего, сейчас бы перед вами не лежала.

ЧАСТЬ I

ПОСТРОЕНИЕ ПРИЛОЖЕНИЙ НА ОСНОВЕ DIRECTX 10

Глава 1

Собираемся в путь

Что такое DirectX и зачем он нужен?

Этот вопрос, наверное, покажется глупым. Каждый ведь знает, для чего нужен DirectX. DirectX — это такая штука, с помощью которой пишутся игры. Все ведь понятно! Однако такой ответ, нужно признать, далек от описания полной картины. Более-менее полное определение выглядит так: DirectX представляет собой набор классов, функций и структур, обеспечивающих высокоэффективное выполнение задач, возникающих при разработке компьютерных игр и мультимедийных приложений. К таким задачам относятся: загрузка и анимация трехмерных моделей, наложение текстур на объемные объекты, воспроизведение звуковых файлов и многие другие. Другими словами, DirectX представляет собой средство разработки приложений, интенсивно использующих мультимедийные возможности компьютера, призванное облегчить жизнь программиста. Если быть до конца точным, DirectX состоит из следующих компонентов.

- **Direct3D** позволяет загружать в память трехмерные модели, осуществлять наложение текстур, рассчитывать освещенность и выводить результат на экран. Именно эта составляющая, отвечающая за вывод двухмерной и трехмерной графики на экран, нас и будет интересовать.
- **DirectSound** отвечает за работу со звуком, его можно использовать при разработке программ для воспроизведения и захвата звукового сигнала.
- **DirectInput** организует работу с устройствами ввода: с клавиатурой, мышью, различного вида джойстиками, включая полную поддержку технологии обратной связи.

Также являются частью DirectX, но считаются устаревшими, следующие компоненты. Многие из них не рекомендуются для разработки новых программ.

- **DirectDraw** присутствовал в DirectX до седьмой версии и отвечал за двухмерную графику. После появления Direct3D необходимость в данном

компоненте отпала, его функции теперь несет на себе Direct3D, хотя возможностями DirectDraw можно пользоваться до сих пор.

- **DirectMusic** отвечает за воспроизведение музыки. Имеется возможность воспроизведения файлов MIDI и WAV. Единственный из перечисляемых компонентов, который не считается устаревшим. Как сказано в документации, "...сохранит свой текущий статус, пока в данной области не появится новая технология".
- **DirectShow** служит для воспроизведения различного рода мультимедийных файлов: MPG, MP3 и других. По сути, он представляет собой программируемый проигрыватель.
- **DirectPlay** позволяет создавать многопользовательские игры, отвечая за обмен данными по сети.

Тем не менее, информацию по использованию всех этих компонентов можно найти в комплекте документации, поставляемой с DirectX SDK, и в базе знаний Microsoft по адресу в Интернете **http://msdn.microsoft.com**.

Немного истории: от Windows 95 к Windows Vista

Трудно представить это сегодня, но сравнительно недавно, чуть больше десяти лет назад, не было ни операционной системы Windows, ни, тем более, каких-либо средств разработки мультимедийных приложений для нее. Строго говоря, Windows существовала, хотя и не в виде операционной системы, а в виде своего рода графической оболочки, запускаемой из-под MS-DOS. В компьютерных изданиях того времени встречались рекомендации не использовать ее якобы из-за того, что простота и доступность графического интерфейса впоследствии затрудняет изучение текстовых команд MS-DOS.

При написании игровых программ в системе MS-DOS программистам приходилось самим реализовывать практически все функции для вывода графики. С одной стороны, это усложняло и затягивало разработку, с другой — можно было пользоваться прямым доступом к оборудованию и "выжимать" из него максимум производительности. Можно было, например, формировать изображение на экране, копируя необходимую информацию прямо в память видеокарты.

С выходом операционной системы Windows 95 ситуация изменилась: приложения в новой системе больше не имели прямого доступа к оборудованию. Вывод графики осуществлялся только с привлечением системных функций, а скорость их работы была далека от той, которая требовалась для разработки

динамичных игр. Это было платой за так называемый *уровень аппаратных абстракций* (HAL — hardware abstraction layer) новой операционной системы, который позволял работать с устройствами компьютера без учета их особенностей в зависимости от фирмы-производителя. Конечно, игры под новую систему были, взять хотя бы набор, поставляемый вместе с Windows. От версии к версии он изменялся мало: "Сапер", пожалуй, самая узнаваемая, "Червы", различные виды карточных пасьянсов. А в MS-DOS, уже "старой" системе, можно было поиграть в "Doom"… Таким образом, сравнение возможностей для разработчиков игр пока было явно не в пользу Windows, требовалось срочно принимать меры. И корпорация Microsoft их приняла. Первая версия библиотеки DirectX (тогда она называлась Windows Games SDK) вышла уже в сентябре 1995 года. В то время, конечно, в составе библиотеки еще не было компонента Direct3D, но и трехмерные ускорители к тому времени еще не получили широкого распространения. Историю Direct3D, наверное, следует начинать с 1992 года, когда в компании RenderMorphics началась разработка трехмерного графического интерфейса прикладного программирования (application programming interface, API) "Reality Lab" для использования в медицине и системах автоматизированного проектирования. В феврале 1995 года компания RenderMorphics была куплена Microsoft, и в результате первая версия API для трехмерной графики была включена во вторую и третью версии библиотеки DirectX. Тем не менее, основная часть разработчиков игр перешла на DirectX лишь с выходом его третьей версии, а с восьмой версии библиотека DirectX стала практически стандартом для разработки компьютерных игр. Примерная хронология выхода версий DirectX представлена в табл. 1.1.

Таблица 1.1. *Хронология выхода версий DirectX*

Версия DirectX	**Операционная система**	**Дата выхода**
DirectX 1.0	Windows 95a	30 сентября 1995
DirectX 2.0 / 2.0a	Windows 95 OSR2 и NT 4.0	5 июня 1996
DirectX 3.0 / 3.0a	Windows NT 4.0 SP3 (последняя версия с поддержкой DirectX для Windows NT 4.0)	15 сентября 1996
DirectX 4.0	-	не выходила
DirectX 5.0	Была доступна в бета-версии под Windows NT 5.0, инсталлировалась под Windows NT 4.0	16 июля 1997
DirectX 5.1	Windows 95/98/NT4.0	1 декабря 1997
DirectX 5.2	Windows 95	5 мая 1998

Таблица 1.1 *(окончание)*

Версия DirectX	Операционная система	Дата выхода
DirectX 5.2	Windows 98	5 мая 1998
DirectX 6.0	Windows 98/NT4.0	7 августа 1998
DirectX 6.1	Windows 95/98/98SE	3 февраля 1999
DirectX 7.0	Windows 95/98/98SE/2000	22 сентября 1999
DirectX 7.0a	Windows 95/98/98SE/2000	сентябрь 1999
DirectX 7.1	Windows 95/98/98SE/ME/2000	16 сентября 1999
DirectX 8.0	Windows 95/98/98SE/ME/2000	30 сентября 2000
DirectX 8.0	Игровая приставка Xbox	3 ноября 2000
DirectX 8.0a	Последняя версия под Windows 95	7 ноября 2000
DirectX 8.1	Windows 98/98SE/ME/2000/XP	12 ноября 2001
DirectX 9.0	Windows Server 2003	19 декабря 2002
DirectX 9.0a	Windows 98/98SE/ME/2000/XP	26 марта 2003
DirectX 9.0b	RC2	13 августа 2003
DirectX 9.0c	Windows XP SP2, Windows Server 2003 SP1, Xbox 360	13 декабря 2004
DirectX 9.0c	Совместимые с DX9.0c версии Windows, впервые включена библиотека D3DX	9 декабря 2005
DirectX 9.0c (Shader Model 3.0)	Windows XP. Последнее обновление с поддержкой Windows 98/98 SE/ME/2000 — в августе 2005	Ежемесячные обновления с августа 2005
DirectX 10.0	Новая версия DirectX (только для Windows Vista)	30 ноября 2006

Сравнение DirectX 9 и DirectX 10

Итак, чем именно новая версия отличается от своего предшественника? Можно выделить пять основных отличий:

1. Расширенные возможности программирования. В десятой версии разработчики полностью отказались от использования фиксированных функций графического конвейера. В новой версии все возлагается на шейдеры: расчет освещения, преобразование координат вершин и т. д.

2. Строгие требования к возможностям оборудования. Любая видеокарта, совместимая с Direct3D 10, обеспечивает один и тот же базовый набор возможностей. Это позволяет отказаться от введения в программу нескольких отдельных ветвей, выполняющихся в зависимости от того, какими возможностями обладает видеокарта.
3. Улучшенная производительность. Снижено количество вызовов API для выполнения часто встречающихся операций.
4. Унифицированная архитектура шейдеров. Это нововведение позволяет более рационально использовать процессор видеокарты, выделяя больше ресурсов для задач, требующих интенсивных вычислений.
5. Геометрические шейдеры и потоковый ввод/вывод. С помощью геометрического шейдера можно на основе данных, переданных с предыдущей стадии, строить новую геометрию. Результат работы геометрического шейдера можно затем вновь передать на вход графического конвейера.

Где достать DirectX 10 SDK?

Ну, вот мы разобрались, что представляет собой DirectX. Пришло время попробовать свои силы в программировании. Очевидно, что без самой этой библиотеки много напрограммировать не получится.

Раньше была традиция записывать текущую версию DirectX SDK на прилагаемый к книге компакт-диск. Но с некоторых пор корпорация Microsoft запретила распространение библиотеки третьими лицами. Поэтому на момент написания книги ее можно только скачать со страницы на сайте Microsoft по следующей ссылке: **http://www.microsoft.com/windows/directx/default.aspx**, размер файла около 450 Мбайт.

> ***Замечание***
>
> DirectX распространяется в двух вариантах: вариант для пользователей и вариант для разработчиков. Нас, конечно же, интересует вариант для разработчиков — DirectX SDK. Сориентироваться можно по объему файла: вариант для пользователей занимает примерно 50 Мбайт, вариант для разработчиков, как уже говорилось — порядка 450 Мбайт.

Установка DirectX 10 SDK на компьютер

Итак, на жестком диске компьютера появился файл, который называется, к примеру, dxsdk_feb2007.exe, если мы скачали версию за февраль 2007 года. Для выполнения установки необходимо произвести следующие действия:

1. Запустить файл dxsdk_feb2007.exe.

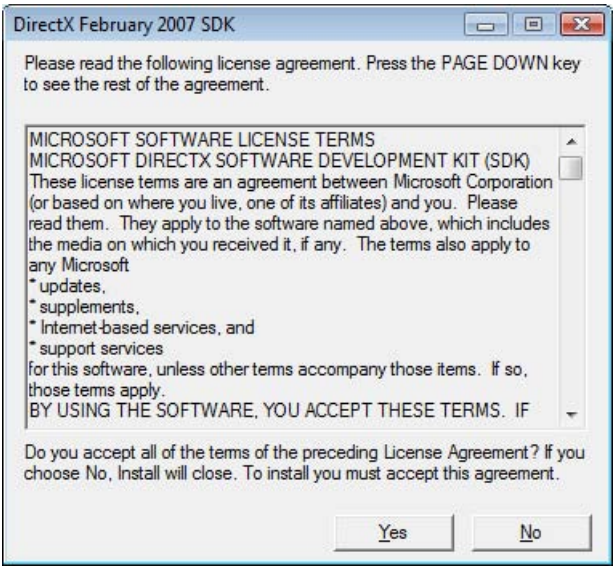

Рис. 1.1. Окно с текстом лицензионного соглашения

2. Появится окно с текстом лицензионного соглашения (рис. 1.1), его необходимо принять (нажать кнопку **Yes** (Да)), чтобы продолжить установку.

> ***Замечание***
>
> Если на компьютере уже установлена какая-либо версия DirectX SDK, перед установкой новой версии ее необходимо удалить.

3. В появившемся окне с заголовком "WinZip Self-Extractor" (рис. 1.2) ввести путь для временного хранения распакованных установочных файлов либо сразу щелкнуть по кнопке **Unzip** (Распаковать).

Рис. 1.2. Окно распаковщика файлов

Рис. 1.3. Окно программы установки

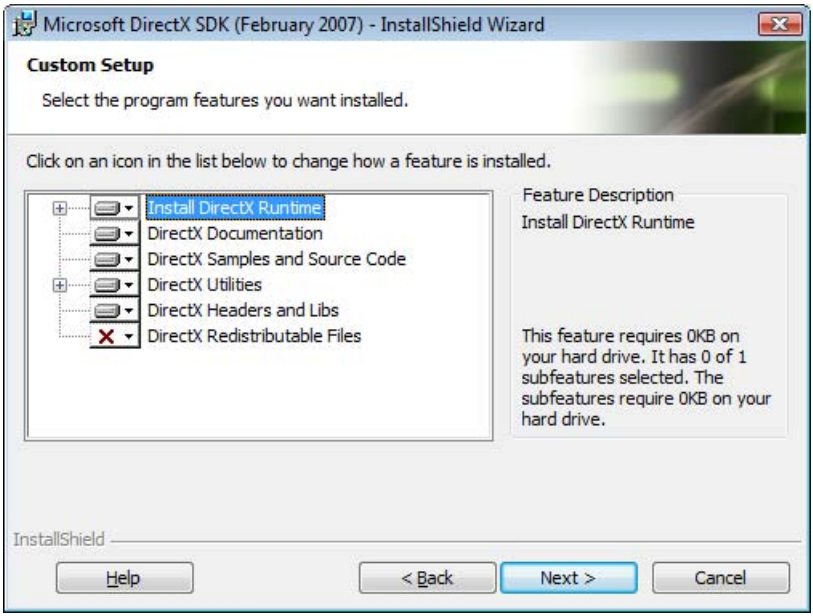

Рис. 1.4. Выбор компонентов DirectX SDK

Замечание

После окончания установки файлы из указанной временной папки автоматически не удаляются, а занимаемое ими пространство на диске составляет около 800 Мбайт. Поэтому лучше либо ввести путь до известной папки (например, C:\Temp), либо скопировать стоящий по умолчанию путь для последующего удаления файлов.

4. После распаковки запустится программа `setup.exe`, в появившемся окне (рис. 1.3) щелкаем на кнопке **Next** (Далее).
5. Для согласия с лицензионным соглашением ставим переключатель в положение "**I accept the terms in the license agreement**" ("Я принимаю условия лицензионного соглашения"), нажимаем кнопку **Next** (Далее).
6. На этом этапе можно выбрать компоненты DirectX SDK для установки (рис. 1.4). Оставляем все как есть, нажимаем кнопку **Next** (Далее).
7. После окончания копирования файлов нажимаем кнопку "**Finish**" ("Завершить"). Все, DirectX SDK установлен!

Среда разработки Visual Studio 2005

Итак, мы установили DirectX SDK на компьютер, но его использование неотделимо от использования среды разработки и компилятора. Рассмотрим работу в среде разработки Visual Studio 2005, в которой мы будем составлять все наши программы. Я думаю, вы уже не раз с ней работали, но, на всякий случай, повторим основные моменты.

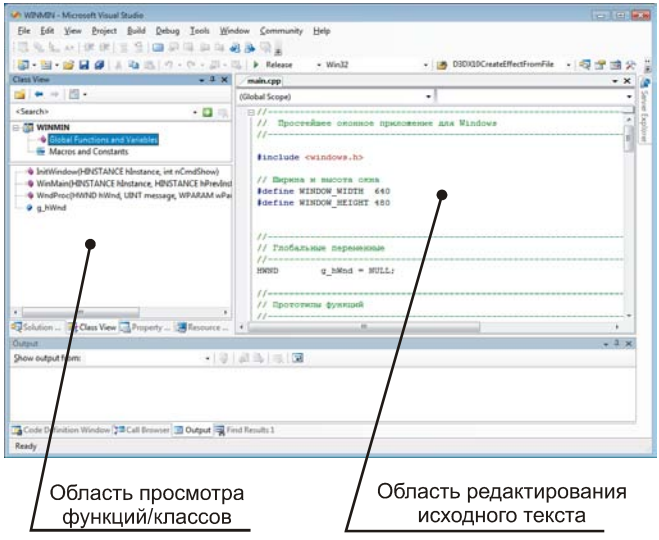

Рис. 1.5. Рабочее окно Visual Studio 2005

Общий вид рабочего окна и элементов интерфейса можно увидеть на рис. 1.5. Для нас будут иметь значение две основные области: область редактирования исходного текста и область просмотра функций/классов (**Class View**). Уже из названия понятно, что и для чего используется.

Создание нового проекта

Рассмотрим создание нового проекта в Visual Studio. В основном меню выбираем: **File | New | Project**… (Файл | Создать | Проект…). В появившемся окне (рис. 1.6) необходимо указать тип создаваемого проекта: в списке **Project types** (Типы проектов) выбираем **Win32**.

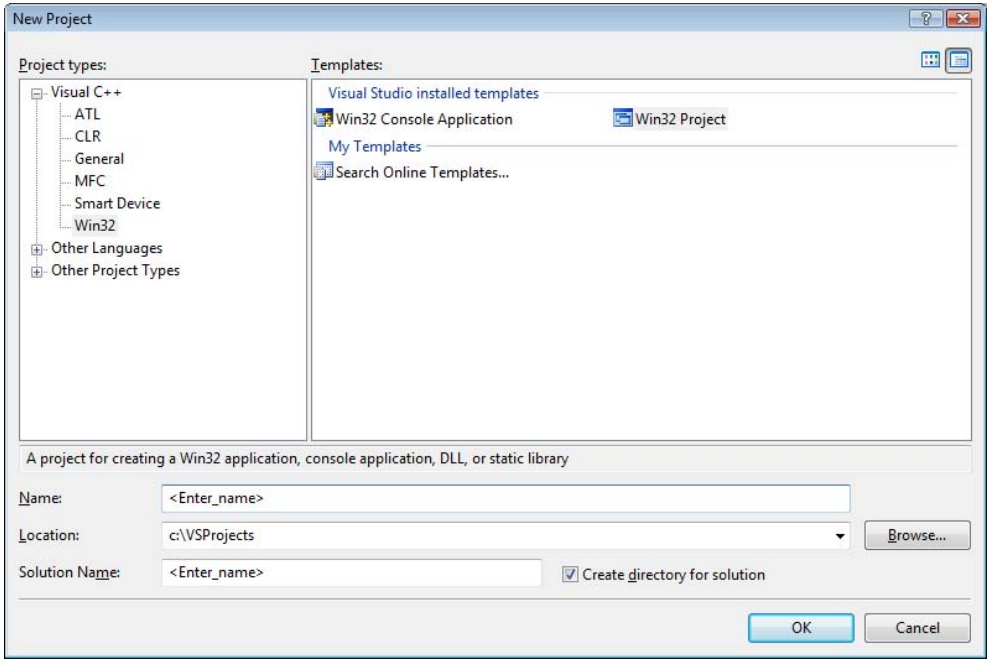

Рис. 1.6. Создание нового проекта в Visual Studio

Также необходимо задать шаблон (то есть какой-то набор настроек, подключаемых библиотек и, быть может, даже готового исходного текста), по которому будет создан наш проект. Для выбора доступны два шаблона: **Win32 Console Application** (Приложение командной строки Win32) и **Win32 Project** (Проект Win32). Нам нужен шаблон **Win32 Project**, поэтому в списке **Templates** (Шаблоны) его и выбираем. В поле **Name** (Наименование) нужно ввести название нашего проекта (здесь нужно придумать что-нибудь впечат-

ляющее, но на всякий случай имейте в виду, что имена Doom и Quake уже заняты). Текст в поле **Name** будет использован как имя для новой папки, в которой будут храниться файлы проекта. В поле **Location** (Расположение) можно указать путь до месторасположения папки с нашим проектом (рис. 1.6). Флажок **"Create directory for solution"** ("Создать директорию для решения") можно снять, на этапе обучения пусть все файлы находятся под рукой, в одной папке. По умолчанию Visual Studio предлагает создать проект в папке "Мои документы" текущего пользователя, где уже содержится папка Visual Studio 2005\Projects. Если такое местонахождение проекта нас устраивает, оставляем все как есть. После того как указаны шаблон для создаваемого проекта, его название и место расположения, нажимаем кнопку **OK**. Нас ожидает еще одно окно, которое предлагает нам настроить дополнительные параметры проекта (рис. 1.7).

Рис. 1.7. Мастер создания приложения Win32

В этом окне нам нужно сделать всего одно изменение: для этого мы переключимся на вкладку **Application Settings** (Параметры приложения), и на ней установим флажок **Empty project** (Пустой проект). Настройки приложения должны выглядеть так, как показано на рис. 1.8.

Рис. 1.8. Настройки нового приложения

Нужно также проверить, что в разделе **Application type** (Тип приложения) переключатель установлен в позицию **Windows application** (Оконное приложение). На этом все подготовительные операции кончаются, и мы наконец-то щелкаем по кнопке **Finish** (Завершить) и получаем новый проект, заготовку для нашей будущей программы.

Добавляем файлы в проект

Проект у нас есть, но он пока что совсем пустой, в нем нет ни строчки исходного текста. Исправить эту ситуацию можно двумя способами: либо мы создаем в проекте новый файл, либо добавляем в него уже имеющийся, например, из ранее созданного проекта. Чтобы добавить файл к проекту, открываем раздел **Project** (Проект) главного меню и в нем выбираем нужный нам пункт: **Add New Item…** (Добавить новый элемент…), если создаем новый элемент, или **Add Existing Item…** (Добавить существующий элемент…), если хотим добавить уже имеющийся. В первом случае откроется новое окно (рис. 1.9), в котором в разделе **Categories** (Категории элементов) мы выбираем пункт **Code** (Текст программы) и затем указываем, какой конкретно файл

нам нужен: с исходным текстом (.cpp) или заголовочный (.h). После всего этого вводим имя нового файла в поле **Name** (Наименование) и щелкаем по кнопке **Add** (Добавить). Во втором случае откроется окно выбора файла, где нужно выбрать файл, который мы хотим добавить в проект.

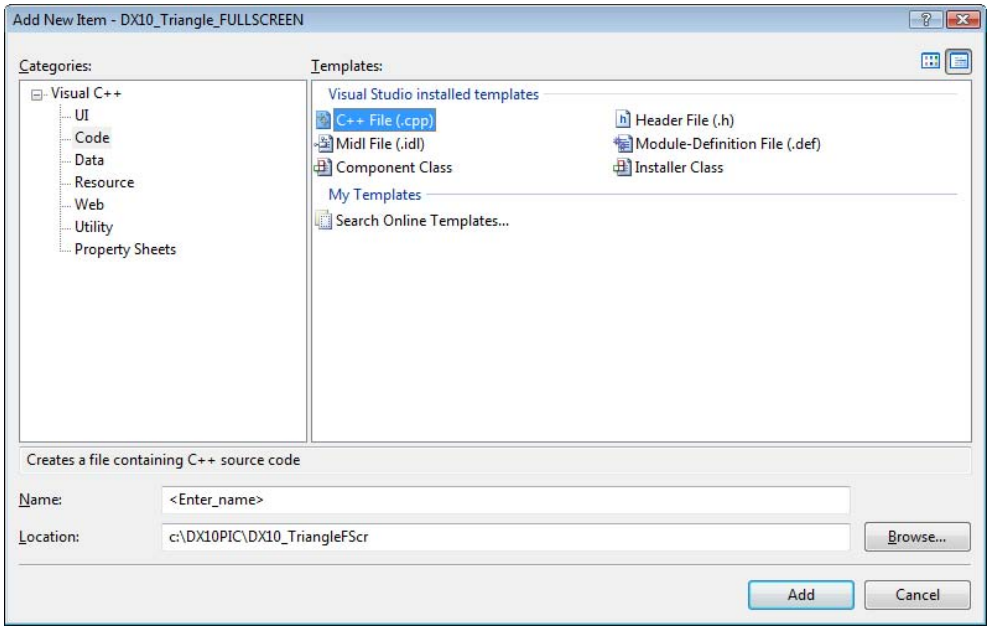

Рис. 1.9. Добавление в проект нового элемента

Настройка путей для компиляции

Если при установке DirectX SDK обнаруживает наличие на компьютере Visual Studio, то пути для всех необходимых файлов программа установки должна добавить автоматически. В случае если этого по каким-то причинам не произошло, пути к файлам необходимо ввести вручную, иначе компиляция программ, использующих DirectX, будет невозможна.

Предположим, мы установили DirectX SDK в папку C:\DXSDK. Как добавить пути в этом случае? Открываем пункт меню **Tools | Options…** (Сервис | Настройки), в открывшемся окне (рис. 1.10) в списке выбираем раздел **Projects and Solutions** (Проекты и решения), в этом разделе нас интересует пункт **VC++ Directories** (Директории VC++). В выпадающем списке **Show directories for** (Отобразить директории для) выбираем **Include files** (Включаемые файлы) и добавляем в список новую строку (щелкаем мышью на кнопке со значком создания новой папки), в которую заносим путь C:\DXSDK\Include.

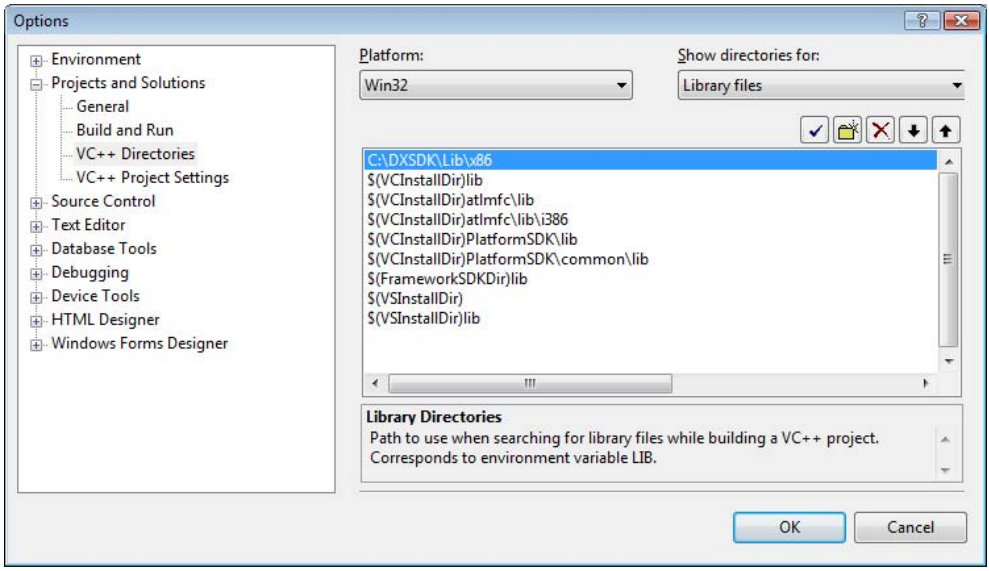

Рис. 1.10. Настройка к путям файлов для компиляции в Visual Studio

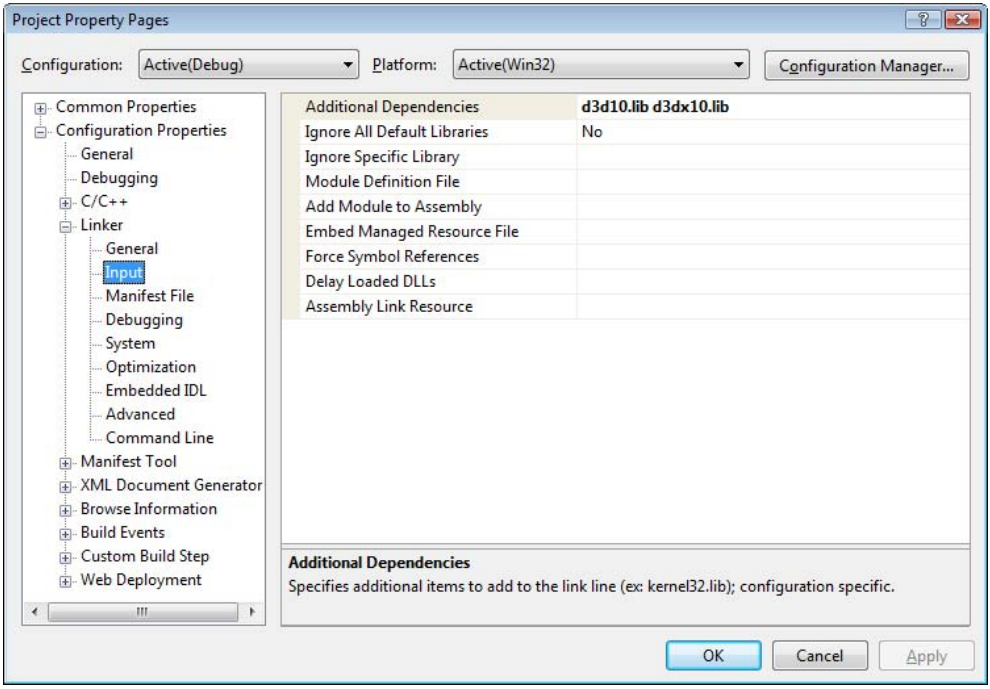

Рис. 1.11. Библиотеки для компиляции программ, использующих Direct3D10

Переключаем список в режим отображения директорий для **Library files** (Библиотечные файлы), добавляем новую строку и пишем в нее следующий путь: C:\DXSDK\Lib\x86, если на компьютере установлен 32-разрядный процессор, и C:\DXSDK\Lib\x64, если процессор 64-разрядный.

Для успешной компиляции программ, использующих Direct3D 10, нужно также предписать использование библиотек d3d10.lib и d3dx10d.lib в свойствах проекта. Сделать это можно таким образом: в главном меню выбираем **Project | Properties** (Проект | Свойства) и в подразделе **Input** (Входные данные) раздела **Linker** (Компоновщик) добавляем в поле **"Additional dependencies"** ("Дополнительные библиотеки") имена файлов библиотек d3d10.lib и d3dx10d.lib (рис. 1.11).

Компиляция и запуск программ

Созданный проект скомпилировать и запустить пока не удастся: мы создали его пустым. Но нужно иметь в виду, что компиляция и запуск программы на выполнение выполняется с помощью кнопки со значком "воспроизведение" (рис. 1.12).

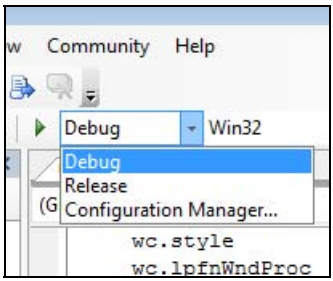

Рис. 1.12. Запуск программы в Visual Studio

Еще, пожалуй, стоит упомянуть, что при отладке и тестовых запусках проекта следует устанавливать в выпадающем списке рядом с кнопкой значение **Debug** (Отладка). Он установлен по умолчанию. При компиляции готовой отлаженной программы "на выпуск" следует установить значение **Release** (Выпуск).

Рабочее место мы подготовили, какой-то минимум информации по работе со средой разработки получили. Для составления программ знаний пока не хватает, но я все постараюсь объяснить по пути. Итак, отправляемся дальше.

Глава 2

Первые шаги

Мы уже умеем создавать новые проекты и компилировать программы в Visual Studio, однако для того, чтобы начать писать программы под Windows, этого еще не достаточно. В программе для Windows должны присутствовать некоторые обязательные элементы, без которых она будет неработоспособна. Чтобы писать программы под DirectX, мы должны уметь создавать простейшее приложение Windows, на "поверхность" окна которого и будет выводиться графика средствами Direct3D. Но прежде чем мы познакомим со структурой приложения Windows, нам необходимо хотя бы в общих чертах представить, как организована операционная система.

Многозадачная операционная система Windows

Начнем с самого начала, с понятия операционной системы. Операционная система (ОС) — это программа, обеспечивающая среду для выполнения приложений и предоставления им доступа к ресурсам компьютера. Под ресурсами понимается оперативная память, процессорное время и вообще любые устройства, которые может использовать приложение во время своей работы.

Что означает слово "многозадачная"? Это значит, что система может обеспечивать одновременное выполнение нескольких приложений. Строго говоря, настоящее параллельное выполнение задач возможно только на процессорах с несколькими ядрами (двух-, четырехъядерные процессоры). При работе на процессорах с одним ядром приложения выполняются по очереди, каждое работает маленький промежуток времени. Промежуток этот носит название кванта времени. За счет того, что длительность кванта времени невелика и достигается иллюзия параллельного выполнения.

В однозадачной системе MS-DOS выполнение программы называлось также "передать программе управление". Это означало, что во время работы программы она монопольно использует все ресурсы компьютера, а чтобы запустить еще одну программу, нужно сначала завершить работу той, которая в данный момент работает. Освобождение ресурсов и передача управления обратно операционной системе происходило по инициативе самой программы. Бывало даже, что если, например, в игровой программе не был предусмотрен выход обратно в систему, то не оставалось ничего другого, как перезагружать компьютер, чтобы завершить ее работу.

> ***Замечание***
>
> На самом деле некоторая возможность параллельного выполнения программ в MS-DOS все же существовала. Имелись так называемые резидентные программы, которые после запуска оставались в оперативной памяти и возвращали управление системе. После этого они могли выполнять какие-либо действия, активизируясь по нажатию "горячей клавиши", даже если одновременно с этим выполнялась какая-то другая программа. С помощью таких программ реализовывалось переключение ввода русских и латинских символов, вывод содержимого графического экрана на принтер либо запись его в файл и т. д.

В многозадачной системе Windows ситуация совершенно другая, приложения выполняются параллельно, одновременно могут работать текстовый редактор, проигрыватель музыкальных файлов, закачиваться данные из Интернета и, вдобавок к этому, выполняться какой-нибудь длительный расчет. Как все это возможно? Как им поделить между собой экран для вывода результатов и разобраться с тем, какому из них адресованы вводимые с клавиатуры символы и которое из них должно реагировать на щелчки мышью? Решение базируется на том, что приложения Windows построены по принципу реакции на события (event-driven application). С точки зрения операционной системы, события представляют собой нажатия на клавиши клавиатуры, движение курсора мыши, а также перемещение, изменение размеров окна приложения и т. д. Приложение не загружает процессор, останавливая всю работу на время ожидания ввода данных, как это происходило в однозадачной системе.

Но как приложение "узнает", например, что ему именно сейчас нужно принимать символы с клавиатуры? Оно получает сообщение от операционной системы. Да, именно так, операционная система Windows управляет приложениями с помощью сообщений. Посылать сообщения могут также и приложения: друг другу и даже сами себе. С каждым окном связана специальная структура — очередь сообщений. В эту структуру и записываются сообщения, предназначенные данному окну. Приложение постоянно проверяет очередь на наличие в ней сообщений. При получении сообщения оно извлекается из очереди и передается на обработку оконной процедуре, специальной

функции того окна, которому предназначено сообщение. Какие действия предпринимать при поступлении того или иного сообщения, определяется именно в этой функции. Фактически в ней содержится оператор `switch`, в качестве параметра которого используется идентификатор поступившего сообщения.

> ***Замечание***
>
> Если быть точным, то очередь сообщений связана с каждым потоком (thread) приложения. Но мы пока будем считать, что она связана с окном, исключительно для простоты объяснения.

Работу оконной процедуры можно понаблюдать следующим образом. Возьмем стандартную программу "Блокнот" и напишем в ней произвольную строку текста. Теперь попробуем выйти из программы, щелкнув мышью на кнопке закрытия окна. Что мы видим? Программа сообщила, что у нас имеется несохраненный текст, и спросила, требуется ли его сохранить. Наша попытка закрыть окно вызвала появление в очереди сообщений программы сообщения `WM_CLOSE` (запрос на закрытие окна). Оно было передано на обработку в оконную процедуру, где и сработала соответствующая ветвь оператора `switch`, в которой предусмотрен запрос на сохранение текста.

Минимальное приложение Windows

Итак, как же должна выглядеть программа под Windows? Структура приложения Windows показана на рис. 2.1.

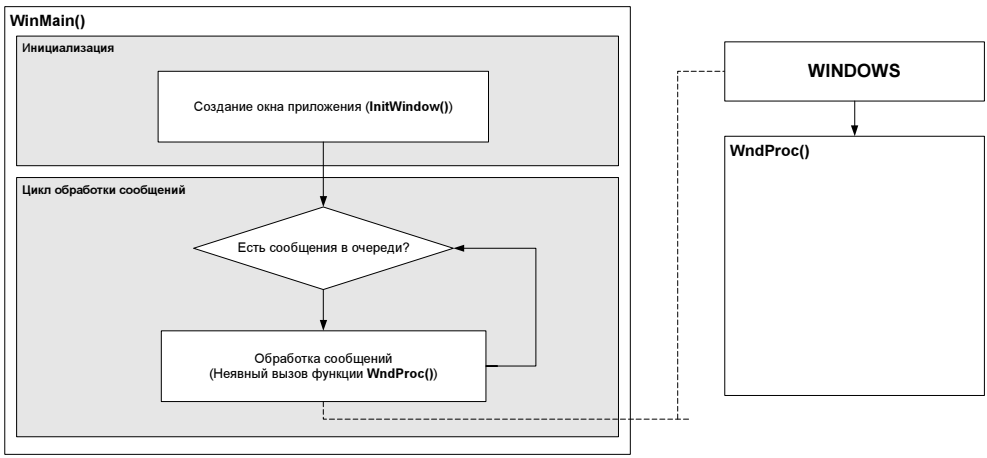

Рис. 2.1. Структура приложения Windows

Как видно из рисунка, приложению Windows для нормальной работы требуется инициализировать окно, создать цикл обработки сообщений и иметь оконную процедуру для соответствующей реакции на них. Выполнение любой программы под Windows начинается с функции `WinMain()`. В этой функции должны производиться следующие действия.

- Инициализация окна приложения: регистрация класса окна (заполнение структуры, определяющей свойства окна приложения, и передача этого набора данных системе под определенным именем — именем класса окна), создание окна по "образцу" — зарегистрированному классу.
- Создание цикла обработки сообщений. В этом цикле сообщения выбираются из очереди и передаются на обработку в оконную процедуру.

> ***Замечание***
>
> Мы рассматриваем только оконные приложения Windows, имеются еще консольные приложения, которые могут не создавать своего окна.

Исходный текст функции `WinMain()` выглядит примерно следующим образом.

```
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow )
{
    if( FAILED( InitWindow( hInstance, nCmdShow ) ) )
        return 0;

    // Цикл обработки сообщений
    MSG msg = {0};
    while( GetMessage( &msg, NULL, 0, 0 ) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }

    return (int) msg.wParam;
}
```

Начнем с самого начала, с объявления функции. Окинем его опытным взглядом. Даже имея лишь приблизительные сведения о функциях в языке C, можно сделать кое-какие выводы. Имя функции, тут без сюрпризов, — WinMain, тип возвращаемого значения — `int`, после него еще вписано какое-то `WINAPI`, функция принимает четыре параметра, какие именно, тоже пока не ясно… Главное мы увидели, теперь разберемся с тем, что пока непонятно.

Модификатор `WINAPI` представляет собой соглашение о вызове функции, оно сообщает компилятору, в какой последовательности должны передаваться параметры функции через стек, а также некоторые другие вещи. Мы подробно останавливаться на этом не будем, для нас это особой роли не играет. Если интересно изучить вопрос глубже, можно заглянуть в заголовочный файл windef.h и посмотреть, что стоит за этим и другими соглашениями. Мы же просто пока примем это как неотъемлемую часть объявления функции `WinMain()`.

Перейдем к параметрам. Они передаются в функцию операционной системой и имеют следующее значение:

- `hInstance` — описатель (handle) приложения в системе (своего рода "регистрационный номер" приложения), очень полезная информация, как мы увидим немного позже;
- `hPrevInstance` — описатель предыдущей запущенной копии приложения, остался от 16-разрядной версии Windows для совместимости, в 32-разрядных версиях всегда равен `NULL`, для нас с вами практической ценности не представляет;
- `lpCmdLine` — указатель на строку с завершающим нулем, которая содержит текст параметров командной строки (без имени исполняемого файла);
- `nCmdShow` — данный параметр определяет состояние окна приложения после запуска, например значение параметра `SW_SHOW` предписывает активировать и отобразить окно на своем текущем положении, значение `SW_MINIMIZE` — свернуть окно на панель задач и т. д.

Мы только что встретили слово "описатель" и с ним нам предстоит столкнуться еще не раз. Что оно означает? Описатель — это внутренний идентификатор Windows, своего рода ссылка на объект в памяти (фактически описатели представляют собой целые числа). Описатель объекта требуется для выполнения каких-либо манипуляций с этим объектом. Например, чтобы указать Direct3D 10, в рабочую область какого окна осуществлять вывод, требуется передать соответствующей функции описатель этого окна. В тексте программы узнать описатель довольно просто: имя такой переменой, как правило, начинается с буквы "h", сокращение от "handle".

Перейдем к телу функции. Здесь мы как раз наблюдаем выполнение тех обязательных действий, которые мы рассмотрели выше. В первой строке происходит вызов функции `InitWindow()`, в которой производится инициализация и создание окна. Сразу после нее создается цикл обработки сообщений. Рассмотрим все по порядку.

Обратите внимание, что вызов функции `InitWindow()` осуществляется из макроса `FAILED`. Здесь все просто: если выражение, заключенное в скобки

возвращает какое-либо значение, свидетельствующее о неудачном завершении функции (код сообщения об ошибке), то макрос возвращает значение TRUE. Нужно сказать, что также имеется аналогичный макрос SUCCEEDED, который выполняет противоположное действие: возвращает TRUE при успешном результате. Данные макросы, как правило, используются для анализа возвращаемых функциями значений. В нашем случае при получении результата TRUE от макроса FAILED (то есть функция InitWindow() завершилась с ошибкой) произойдет выход из программы. Сама функция InitWindow() выглядит так:

```
HRESULT InitWindow( HINSTANCE hInstance, int nCmdShow )
{
    // Регистрируем класс окна
    WNDCLASSEX wc;
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style          = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc    = WndProc;
    wc.cbClsExtra     = 0;
    wc.cbWndExtra     = 0;
    wc.hInstance      = hInstance;
    wc.hIcon          = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground  = (HBRUSH)COLOR_WINDOW;
    wc.lpszMenuName   = NULL;
    wc.lpszClassName  = L"SimpleWindowClass";
    wc.hIconSm        = LoadIcon(NULL, IDI_APPLICATION);
    if( !RegisterClassEx(&wc) )
        return E_FAIL;

    // Создаем окно
    g_hWnd = CreateWindow(
             L"SimpleWindowClass",
             L"Простейшее приложение для Windows",
             WS_OVERLAPPEDWINDOW,
             CW_USEDEFAULT,
             CW_USEDEFAULT,
             640,
             480,
             NULL,
             NULL,
```

```
            hInstance,
            NULL);

  //Если не удалось создать окно - выходим из функции
    if( !g_hWnd )
        return E_FAIL;
        //Отображаем окно на экране
    ShowWindow( g_hWnd, nCmdShow );
    UpdateWindow(g_hWnd);

    return S_OK;
}
```

Итак, взглянем на объявление функции `InitWindow()`. Недоумение вызывает тип возвращаемого значения, `HRESULT`. На самом деле ничего страшного здесь нет, тип `HRESULT` — это тот же тип `LONG`, но он используется для предоставления расширенной информации о результате завершения функции. В функции `InitWindow()` мы будем использовать только два "расширенных" значения: `S_OK` и `E_FAIL`. Нетрудно догадаться, что результат `S_OK` функция вернет в случае успешного завершения, а `E_FAIL` — в случае, если в процессе выполнения произошла ошибка. Конечно же, можно было бы с таким же успехом использовать в качестве типа возвращаемого значения функции тип `BOOL`, но мы собираемся работать с Direct3D 10, а там практически все методы возвращают значения типа `HRESULT`. Будем сразу привыкать к работе с этим типом.

Теперь обратимся к параметрам функции, здесь их у нас всего два, и с их помощью мы передаем в функцию `InitWindow()` значения из функции `WinMain()`. Напомню, первый — это описатель экземпляра приложения, второй — определяет состояние окна приложения после запуска. Теперь посмотрим, что же у этой функции внутри.

Сначала мы регистрируем класс окна с помощью функции `RegisterClassEx()`. Что такое класс окна? Это структура типа `WNDCLASSEX`, содержащая значения параметров, определяющих свойства окна, и эту структуру мы регистрируем в системе. Класс окна является своего рода шаблоном, позволяющим быстро создать окно нужного типа. Чтобы лучше понимать, о чем идет речь, вспомним, как выглядит окно приложения в системе Windows (рис. 2.2).

Можно выделить заголовок окна, в котором отображается текст, как правило, название программы, меню, с помощью которого осуществляется выполнение различных действий, клиентскую область, куда приложение выводит информацию (может "рисовать" в этих пределах), а также рамку, границу окна.

Инициализация как раз и заключается в описании параметров этих элементов. Часть из них мы определяем с помощью класса окна.

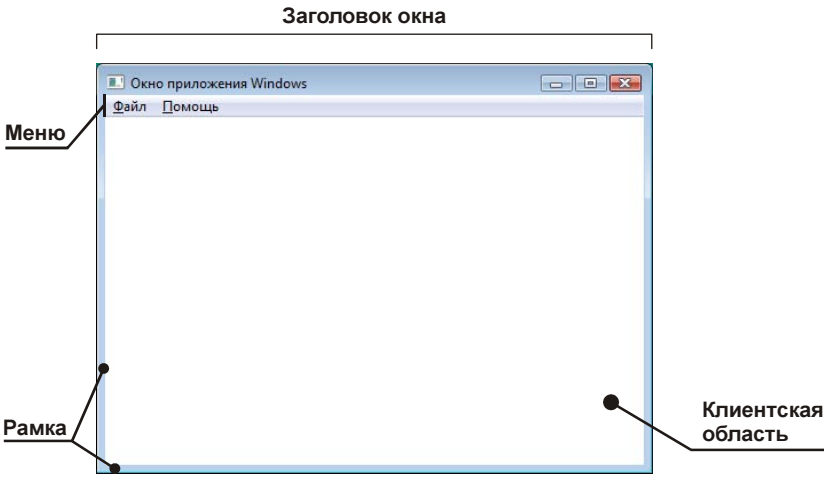

Рис. 2.2. Окно приложения Windows

Зарегистрированный класс мы используем на следующем этапе при создании окна приложения с помощью функции `CreateWindow()`, а пока разберемся, как заполняется структура `WNDCLASSEX`, и какую информацию она содержит. Определение структуры `WNDCLASSEX` представлено ниже:

```
typedef struct {
    UINT cbSize; // Размер структуры в байтах
    UINT style;  // Стиль окна
    WNDPROC lpfnWndProc; // Указатель на оконную процедуру
    int cbClsExtra; // Резервирование памяти для
    int cbWndExtra; //дополнительной информации
    HINSTANCE hInstance; // Описатель приложения
    HICON hIcon; // Описатель основного значка
    HCURSOR hCursor; // Описатель курсора окна
    HBRUSH hbrBackground; // Кисть для фона окна
    LPCTSTR lpszMenuName; // Имя ресурса меню
    LPCTSTR lpszClassName; // Имя класса
    HICON hIconSm; // Описатель малого значка
} WNDCLASSEX, *PWNDCLASSEX;
```

Сначала мы объявляем переменную типа `WNDCLASSEX`:

```
WNDCLASSEX wc;
```

Следующим шагом мы заполняем первое поле, размер структуры в байтах:

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

Наверное, выглядит это немного странно: структура, содержащая размер о самой себе. Однако представим, что размер структуры понадобится для работы каким-нибудь функциям. Они смогут сразу получить его, просто прочитав поле структуры, не тратя время на вычисления. Если иметь это в виду, хранение размера структуры выглядит вполне логично.

Далее следует параметр, содержащий битовые флаги, определяющие стиль окна (общие свойства). В этой строке мы указываем на необходимость перерисовки всего окна при изменении его горизонтальных или вертикальных размеров (комбинация стилей `CS_HREDRAW` и `CS_VREDRAW` соответственно; флаги стилей можно комбинировать между собой с помощью битового оператора OR ("`|`")).

```
wc.style = CS_HREDRAW | CS_VREDRAW;
```

В следующей строке мы вписываем указатель на функцию, которая будет обрабатывать сообщения для окна описываемого класса, то есть на оконную процедуру нашего приложения. Впишем имя функции `WndProc` и запомним его: именно так мы должны назвать нашу оконную процедуру.

```
wc.lpfnWndProc = WndProc;
```

Следующие два поля, `cbClsExtra` и `cbWndExtra`, позволяют программисту выделить дополнительное место в памяти под произвольные данные. Обычно эти поля заполняют нулями, так же поступим и мы.

```
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
```

В поле `hInstance` требуется занести описатель экземпляра приложения, в котором находится оконная процедура для окна описываемого класса. Сюда мы просто заносим значение, которое система Windows передала функции `WinMain()` при запуске программы:

```
wc.hInstance = hInstance;
```

Далее следует поле, содержащее информацию о значке приложения, точнее — его описатель. Для нашего первого приложения нас вполне устроит системный значок по умолчанию. Чтобы установить его, используем для загрузки значка функцию `LoadIcon()`:

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

Следующее поле играет аналогичную роль: оно содержит описатель курсора, который следует отображать, когда указатель мыши находится над клиентской областью окна. Ничего особенного нам не требуется, так что установим

обычную стрелку по умолчанию. Для загрузки изображения курсора также существует своя функция под именем LoadCursor():

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

В поле hbrBackground требуется ввести описатель кисти, попросту говоря цвет, которым будет рисоваться клиентская область окна. Установим белый цвет фона (стандартный цвет окна, начиная с Windows XP), для этого просто приведем константу COLOR_WINDOW к типу HBRUSH:

```
wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
```

> ***Замечание***
>
> В Windows XP и Windows Vista стандартный цвет окна — белый. В других версиях Windows цвет окна по умолчанию может отличаться, например в Windows 95 окно отобразится с серым цветом фона. Вы наверняка обратили внимание, что приводим мы к типу HBRUSH значение на единицу большее: (COLOR_WINDOW+1). Это вызвано тем, что значения индексов цветов начинаются с нуля, а описателей кисти — с единицы.

Следующее поле представляет собой строку с завершающем нулем, в которой содержится имя ресурса меню. Мы с меню работать не собираемся, поэтому смело можно записывать туда NULL:

```
wc.lpszMenuName    = NULL;
```

В следующее поле запишем имя описываемого нами класса. Назовем его SimpleWindowClass — "простой класс окна". Обратите внимание, что строковые константы необходимо записывать с литерой "L" перед кавычками, так, как это требуется для работы с символами UNICODE:

```
wc.lpszClassName = L"SimpleWindowClass";
```

Теперь мы сможем обращаться из программы к нашему классу по его имени. У нас осталось последнее поле, в которое мы поместим описатель малого значка. Это значок, который выводится в заголовке окна и на панели задач Windows. Воспользуемся все той же функцией LoadIcon() и установим значок по умолчанию:

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

Вот мы и заполнили все поля структуры WNDCLASSEX. Теперь, для того чтобы получить возможность создавать окна с помощью этого класса, мы должны его зарегистрировать в системе. Сделать это можно с помощью функции RegisterClassEx(). Функция принимает один-единственный параметр — указатель на переменную типа WNDCLASSEX и возвращает TRUE, если функция успешно завершилась и FALSE, если возникли ошибки. Исходя из этого, вызов функции мы производим следующим образом:

```
if( !RegisterClassEx(&wc) )
        return E_FAIL;
```

Функции передается адрес переменной `wc`, при неудачном завершении происходит выход из функции `InitWindow()` с возвращением значения `E_FAIL`.

Часть функции, ответственную за описание и регистрацию класса окна, мы рассмотрели. Теперь рассмотрим оставшуюся часть, в которой происходит создание окна с использованием зарегистрированного класса.

Для начала посмотрим, что представляет собой прототип функции `CreateWindow()`:

```
HWND CreateWindow(
    LPCTSTR lpClassName,// Указатель на имя класса
    LPCTSTR lpWindowName,// Указатель на текст в заголовке окна
    DWORD dwStyle, // Внешний вид окна (стиль)
    int x, // Координаты расположения
    int y, // окна на экране
    int nWidth, // Ширина окна
    int nHeight, // Высота окна
    HWND hWndParent, // Описатель родительского окна
    HMENU hMenu, // Описатель меню или дочернего окна
    HINSTANCE hInstance, // Описатель экземпляра приложения
    LPVOID lpParam // Указатель на данные создания окна
);
```

В случае своего успешного завершения функция возвращает описатель вновь созданного окна; в противном случае она возвращает значение `NULL`. Вызываем мы функцию `CreateWindow()` аналогично тому, как вызывали функцию `RegisterClassEx()`, с той лишь разницей, что выход из функции `InitWindow()` происходит при возвращении функцией значения `NULL`. Назначение параметров функции в основном ясно из комментариев, однако остановимся на каждом в отдельности:

- `lpClassName` — здесь мы указываем имя класса, который только что зарегистрировали (`L"SimpleWindowClass"`);
- `lpWindowName` — этот параметр определяет текст в заголовке создаваемого окна, запишем его таким образом: `L"Простейшее приложение Windows"`;
- `dwStyle` — флаги, определяющие внешний вид и поведение окна, здесь мы указываем только одно значение: `WS_OVERLAPPEDWINDOW` (в табл. 2.1 можно познакомиться и с некоторыми другими значениями);
- `x` и `y` — координаты верхнего левого угла окна в пикселах на момент создания. И для `x`, и для `y` мы указываем значение `CW_USEDEFAULT`, оно означает, что координаты окна система установит сама;
- `nWidth` и `nHeight` — определяют ширину и высоту окна в пикселах, для которых мы устанавливаем значения `640` и `480`, соответственно;

- `hWndParent` — описатель родительского окна, здесь мы указываем значение `NULL`, обозначая этим, что родительское окно в нашем случае — поверхность рабочего стола;
- `hMenu` — описатель меню, но как мы уже говорили, меню мы не используем, поэтому также пишем `NULL`;
- `hInstance` — описатель экземпляра приложения, мы уже использовали его при заполнении структуры `WNDCLASSEX`, и здесь мы указываем то же значение: `hInstance`;
- `lpParam` — последний параметр, его устанавливаем в `NULL`.

Таблица 2.1. *Некоторые значения параметра dwStyle*

Значение	Описание
`WS_MINIMIZE`	Создает первоначально минимизированное окно
`WS_MAXIMIZE`	Создает первоначально максимизированное окно
`WS_BORDER`	Создает окно с тонкой границей
`WS_CAPTION`	Создает окно, у которого имеется заголовок (уже включает в себя стиль `WS_BORDER`)
`WS_SYSMENU`	Создает окно, у которого имеется оконное меню в заголовке (используется совместно со стилем `WS_CAPTION`)
`WS_MAXIMIZEBOX`	Создает окно, у которого имеется кнопка "Развернуть" в заголовке (используется совместно со стилем `WS_SYSMENU`)
`WS_MINIMIZEBOX`	Создает окно, у которого имеется кнопка "Свернуть" в заголовке (используется совместно со стилем `WS_SYSMENU`)
`WS_THICKFRAME`	Создает окно, имеющее изменяемую границу
`WS_OVERLAPPED`	Создает перекрывающееся окно (у перекрывающегося окна имеется граница и заголовок)
`WS_OVERLAPPEDWINDOW`	Создает перекрывающееся окно, включает в себя стили `WS_OVERLAPPED`, `WS_CAPTION`, `WS_SYSMENU`, `WS_THICKFRAME`, `WS_MAXIMIZEBOX` и `WS_MINIMIZEBOX`

В случае успешного завершения функции мы получим описатель созданного окна и сохраним его в переменной `g_hWnd`, он нам еще пригодится. После выполнения функции `CreateWindow()` окно уже создано, но на экране его пока не видно. Чтобы операционная система его отобразила, воспользуемся функцией `ShowWindow()` для установки состояния окна после запуска. Помните

про второй параметр функции? Сейчас мы его и используем. Прототип функции `ShowWindow()`:

```
BOOL ShowWindow(
    HWND hWnd, // Описатель окна
    int nCmdShow // Состояние окна
);
```

Функция возвращает `TRUE`, если окно было видимым до выполнения функции, и `FALSE`, если оно было невидимо. Для вызова функции требуется указать два параметра: описатель окна, к которому применяется функция, и состояние окна, которое функция должна установить. Описатель у нас есть, состояние окна функция `InitWindow()` получила в качестве параметра `nCmdShow`. Вызовем функцию:

```
ShowWindow( g_hWnd, nCmdShow );
```

В качестве последнего штриха вызовем еще одну функцию, функцию `UpdateWindow()`. Эта функция, если необходимо, посылает окну сообщение, которое является сигналом к перерисовке клиентской области окна (сообщение `WM_PAINT`).

```
UpdateWindow( g_hWnd );
```

После этого мы можем быть уверены, что наше окно правильно отображается на экране. Пора выходить из функции `InitWindow()`, она свою работу сделала. Последняя строка функции возвращает код успешного завершения (`S_OK`):

```
return S_OK;
```

По нашему плану создания приложения Windows мы выполнили первый пункт: инициализировали и создали окно. Функция `InitWindow()` завершила свою работу, и выполнение программы продолжается в функции `WinMain()`. Строк в ней осталось совсем немного, и они как раз представляют собой цикл обработки сообщений. Как мы знаем, в цикле происходит извлечение сообщения из очереди и отправка его на обработку. Рассмотрим цикл отдельно:

```
// Цикл обработки сообщений
MSG msg = {0};
while( GetMessage( &msg, NULL, 0, 0 ) )
{
    TranslateMessage( &msg );
    DispatchMessage( &msg );
}
```

Что происходит в этом фрагменте программы? В первой строке объявляется переменная типа `MSG` (этот тип представляет собой структуру данных, мы ее

пока не раскрываем) и инициализируется нулевыми значениями. Далее мы видим цикл `while`, который будет выполняться до тех пор, пока функция `GetMessage()` возвращает `TRUE`. В теле цикла две функции: `TranslateMessage()` и `DispatchMessage()`.

Сначала опишем функцию `GetMessage()`, это именно она извлекает сообщение из очереди. При наличии сообщений в очереди функция извлекает первое из них по порядку, если сообщения в очереди отсутствуют, функция ожидает их поступления. В случае если функция извлекает сообщение `WM_QUIT`, она возвращает нулевое значение (оператор `if` интерпретирует его как `FALSE`), в любом другом случае функция возвращает ненулевое значение (`TRUE`). Параметров у функции четыре, но мы задаем только один, остальные три обнуляем. Взглянем на прототип функции `GetMessage()`:

```
BOOL GetMessage(
    LPMSG lpMsg, // Указатель на структуру MSG
    HWND hWnd, // Описатель окна
    UINT wMsgFilterMin, // Нижний предел фильтрации
    UINT wMsgFilterMax  // Верхний предел фильтрации
);
```

Теперь отдельно рассмотрим каждый параметр.

- `lpMsg` — указатель на структуру `MSG`, в которую помещается информация об извлеченном из очереди сообщении.
- `hWnd` — описатель окна, которому адресуются сообщения, функция будет извлекать из очереди сообщения, адресованные только этому окну. В нашем случае мы указали значение `NULL`, это означает, что будут извлекаться все сообщения для окон, принадлежащих вызывающему функцию потоку. У нас только одно окно, поэтому это означает просто — "извлекать все сообщения".
- `wMsgFilterMin` и `wMsgFilterMax` — параметры, позволяющие организовать фильтрацию сообщений, которые извлекает функция. Указанные значения будут использоваться в качестве предельных: сообщения с кодом меньше, чем значение `wMsgFilterMin`, и больше, чем `wMsgFilterMax`, будут игнорироваться. Мы задаем нулевые значения для обоих параметров. При этом фильтрация осуществляться не будет, функция будет извлекать все имеющиеся сообщения.

Итак, сообщение из очереди мы извлекли и перешли в тело цикла. Здесь у нас две функции, каждая из которых в качестве параметра принимает указатель на структуру `MSG` с данными сообщения. Первая из них, `TranslateMessage()`, выполняет преобразование кодов виртуальных клавиш к символам ASCII.

Наверное, не слишком понятно, но делает она именно это. Вдаваться в подробности не будем, а сразу перейдем к функции `DispatchMessage()`. Эта функция отправляет извлеченное сообщение на обработку в оконную процедуру. Прежде чем мы приступим к рассмотрению оконной процедуры, давайте выясним, что представляет собой структура `MSG`. Выглядит она следующим образом:

```
typedef struct MSG{
    HWND hWnd;
    UINT message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD time;
    POINT pt;
};
```

Какая информация содержится в структуре сообщения?

- `hWnd` — содержит описатель окна, оконной процедуре которого адресовано сообщение.
- `message` — содержит идентификатор сообщения, то есть его тип (например: `WM_QUIT`, `WM_PAINT` и т. д.).
- `wParam` и `lParam` — содержат дополнительную информацию о сообщении, что именно она собой представляет, зависит от типа сообщения.
- `time` — указывает время, в которое сообщение отправлено.
- `pt` — содержит координаты курсора мыши в момент отправки сообщения (в системе координат экрана).

Теперь посмотрим, какие данные передаются оконной процедуре и как она работает. Полностью текст оконной процедуры приведен ниже.

```
LRESULT CALLBACK WndProc( HWND hWnd, UINT message, WPARAM wParam,
                          LPARAM lParam )
{
  switch (message)
  {

      case WM_DESTROY:
          PostQuitMessage(0);
          break;

      default:
```

```
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

ЗАМЕЧАНИЕ

Оконную процедуру мы назвали `WndProc()`, так как это имя мы указали при описании класса окна. Вы можете использовать любое допустимое имя функции, не забывайте только соответственно указывать его в поле `lpfnWndProc` структуры `WNDCLASSEX` при регистрации класса.

Сразу обратим внимание на модификатор `CALLBACK`, с его помощью мы указываем, что `WndProc()` является функцией обратного вызова. Не нужно пугаться этого названия, оно всего лишь означает, что функция вызывается системой как реакция на какое-либо событие, а не из нашей программы напрямую (в нашем случае событием считается извлечение сообщения из очереди).

ЗАМЕЧАНИЕ

Если сравнить определения для `WINAPI` и `CALLBACK` в файле WinDef.h, можно увидеть, что они представляют собой одно и то же соглашение о вызове функций: `__stdcall`. Таким образом, эти модификаторы взаимозаменяемы, дело лишь в понятности исходного текста.

Как можно заметить, в оконную процедуру передаются не все данные, которые содержит структура `MSG`, заполненная функцией `GetMessage()`. Функция `WndProc()` получает описатель окна, которому адресовано сообщение (`hWnd`), идентификатор сообщения (`message`) и его параметры (`wParam` и `lParam`). Оконная процедура может либо сама обработать поступившее сообщение, либо вызвать системную функцию `DefWindowProc()`, которая проведет обработку сообщения по умолчанию. Писать обработчики для всех имеющихся сообщений Windows — задача, пожалуй, невыполнимая. Поэтому просто необходимо для необрабатываемых сообщений вызывать функцию `DefWindowProc()` и возвращать результат ее работы, так как он может понадобиться операционной системе. Перейдем к телу функции, Здесь у нас имеется оператор `switch`, который в зависимости от поступившего идентификатора сообщения выполняет те или иные операторы. В приведенной здесь оконной процедуре осуществляется обработка только одного сообщения, имеющего идентификатор `WM_DESTROY`. Это сообщение система посылает в оконную процедуру сразу после уничтожения окна (например, если окно закрыто щелчком мыши на кнопке "Закрыть" в заголовке окна). При получении этого сообщения оконная процедура должна вызвать функцию освобождения всех занятых ресурсов и поместить в очередь сообщений приложения

сообщение `WM_QUIT`. Последнее необходимо для того, чтобы приложение завершило свою работу. Тот факт, что приложение посылает сообщение самому себе, нас смущать никоим образом не должен. Функция `GetMessage()` извлечет его из очереди и вернет нулевое значение, что, в свою очередь, вызовет окончание цикла `while` обработки сообщений. За циклом обработки сообщений следует оператор `return`, он производит выход из функции `WinMain()` и, соответственно, завершение работы приложения. Нужно понимать, что если не поместить в очередь сообщение `WM_QUIT` после уничтожения окна, то приложение останется в памяти, занимая ресурсы системы. Так как отправка сообщения `WM_QUIT` — операция очень распространенная, для нее имеется специальная функция: `PostQuitMessage()`, которую и используем. Прототип функции выглядит так:

```
void PostQuitMessage(
    int nExitCode
);
```

Функция не возвращает никакого значения и принимает единственный параметр — код завершения программы, это значение передается в параметре `wParam` сообщения `WM_QUIT` и возвращается операционной системе оператором `return` при завершении функции `WinMain()`.

Кратко подведем итоги, чтобы все окончательно утряслось и сложилось в цельную картину.

- Выполнение приложения для Windows начинается с функции `WinMain()`. В этой функции мы инициализируем окно (создаем и регистрируем класс окна и создаем окно данного класса) и организуем цикл обработки сообщений (в нем выполняется извлечение сообщений из очереди и отправка на обработку в оконную процедуру).
- Обработка сообщений осуществляется в оконной процедуре — функции `WndProc()`, это функция обратного вызова, вызывается системой. Внутри функции имеется оператор `switch`, который в зависимости от идентификатора сообщения выполняет различные операторы. Для необрабатываемых сообщений производится вызов функции обработки сообщений по умолчанию и возвращается результат ее выполнения.

Исходный текст минимального приложения для Windows, которое мы создавали с начала этой главы, представлен в листинге 2.1 (соответствующий проект можно найти на прилагаемом компакт-диске в директории Glava2\WinMin). Необходимо сделать некоторые пояснения к листингу.

В первой строке программы мы подключаем заголовочный файл windows.h, он необходим, чтобы компилировались все функции Windows, которые мы используем в программе. Сразу после этого мы определяем константы для

высоты и ширины окна, на случай, если нам захочется задать другие размеры окна, чтобы не рыскать по всему исходному тексту. За определением констант следует определение глобальной переменной, в которой мы будем хранить описатель окна нашего приложения, так нам будет удобнее. После этого идут прототипы функций, а затем и сами функции нашего приложения. Результат работы приложения можно увидеть на рис. 2.3.

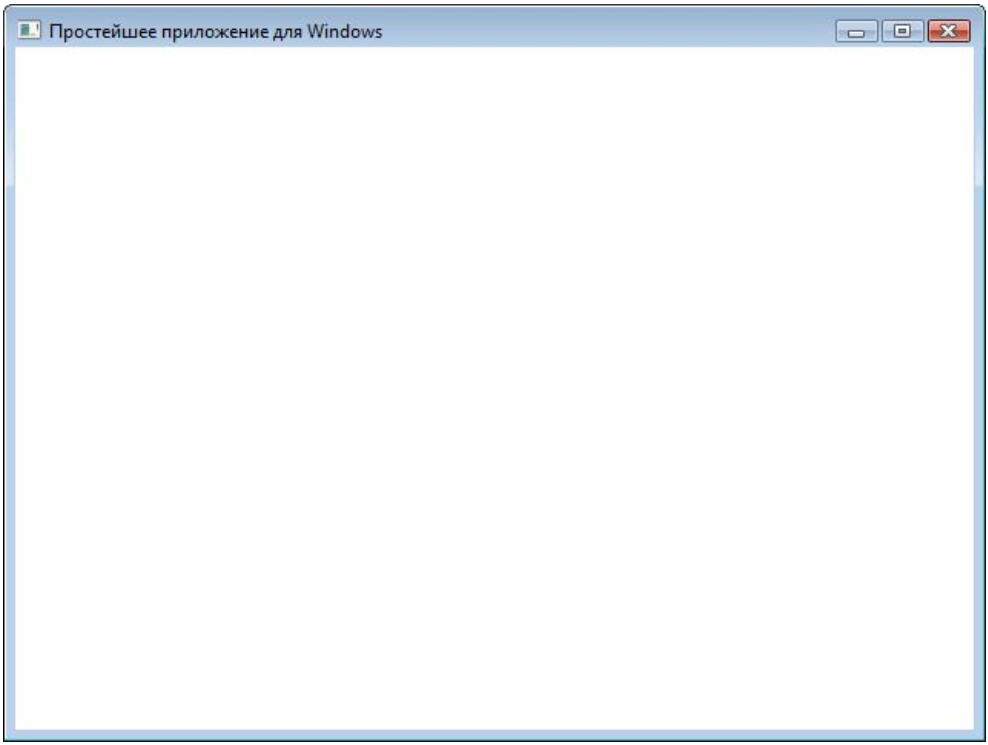

Рис. 2.3. Минимальное приложение Windows

Итак, мы кратко познакомились с функционированием операционной системы Windows и написали минимальное оконное приложение. Сделав этот шаг, мы уже вполне можем сделать и следующий — создание минимального приложения Direct3D 10.

Листинг 2.1

```
//------------------------------------------------------------
// Простейшее оконное приложение для Windows
//------------------------------------------------------------
```

```
#include <windows.h>

// Ширина и высота окна
#define WINDOW_WIDTH  640
#define WINDOW_HEIGHT 480

//--------------------------------------------------------
// Глобальные переменные
//--------------------------------------------------------
HWND        g_hWnd = NULL;
//--------------------------------------------------------
// Прототипы функций
//--------------------------------------------------------
HRESULT             InitWindow( HINSTANCE hInstance,
int nCmdShow );
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);

//--------------------------------------------------------
// С этой функции начинается выполнение программы
//--------------------------------------------------------
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow )
{
    if( FAILED( InitWindow( hInstance, nCmdShow ) ) )
        return 0;

    // Цикл обработки сообщений
    MSG msg = {0};
    while( GetMessage( &msg, NULL, 0, 0 ) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }

    return (int) msg.wParam;
}

//--------------------------------------------------------
// Регистрация класса и создание окна
//--------------------------------------------------------
```

```
HRESULT InitWindow( HINSTANCE hInstance, int nCmdShow )
{
    // Регистрируем класс окна
    WNDCLASSEX wc;
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style          = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc    = WndProc;
    wc.cbClsExtra     = 0;
    wc.cbWndExtra     = 0;
    wc.hInstance      = hInstance;
    wc.hIcon          = NULL;
    wc.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszMenuName   = NULL;
    wc.lpszClassName  = L"SimpleWindowClass";
    wc.hIconSm        = LoadIcon(NULL, IDI_APPLICATION);
    if( !RegisterClassEx(&wc) )
        return E_FAIL;

    // Создаем окно
    g_hWnd = CreateWindow(
            L"SimpleWindowClass",
            L"Простейшее приложение для Windows",
            WS_OVERLAPPEDWINDOW,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            WINDOW_WIDTH,
            WINDOW_HEIGHT,
            NULL,
            NULL,
            hInstance,
            NULL);

    // Если не удалось создать окно - выходим из функции
    if( !g_hWnd )
        return E_FAIL;
    // Отображаем окно на экране
    ShowWindow( g_hWnd, nCmdShow );
```

```
    UpdateWindow(g_hWnd);

    return S_OK;
}

//--------------------------------------------------------------
// Обработка сообщений
//--------------------------------------------------------------
LRESULT CALLBACK WndProc( HWND hWnd, UINT message, WPARAM wParam,
                          LPARAM lParam )
{
    switch (message)
    {
        case WM_DESTROY:
            PostQuitMessage(0);
            break;

        default:
            return DefWindowProc(hWnd, message,
                                 wParam, lParam);
    }

    return 0;
}
```

Минимальное приложение Direct3D 10

Для нашего первого приложения Direct3D воспользуемся только что созданным минимальным приложением для Windows, расширив его функциональность. Приступим. Создадим новый проект и скопируем в него весь код из предыдущего проекта. Однако, прежде чем мы продолжим, нам необходимо еще кое-что узнать.

Технология COM

Как мы выяснили в *гл. 1*, библиотека DirectX представляет собой набор классов, функций и структур, сейчас еще добавим — реализована она с помощью технологии COM. COM (Component Object Model) — "модель составных объектов", специальный стандарт, описывающий правила создания и взаимо-

действия объектов в среде Windows. Что это нам дает? В общих чертах, это предоставляет нам пользоваться функциональными возможностями других приложений Windows, совсем не обязательно написанных нами. Применительно же к DirectX это означает, что библиотека имеет обратную совместимость, и она не зависима от языка программирования вызывающего приложения. Другими словами, мы можем использовать одну и ту же библиотеку в программах на языке C++, Delphi и даже Visual Basic. Как это реализовано? С помощью специальной функции библиотеки создается COM-объект, его обычно называют интерфейсом. В использовании он похож на обычный класс C++. Отличие состоит в том, что мы получаем указатель на интерфейс при помощи специальной функции либо метода другого COM-объекта, а не используем для его создания оператор `new`. Аналогично, при освобождении ресурсов мы освобождаем объект, вызывая его собственный метод `Release`, а не используя оператор `delete`. Самое приятное, что нам совсем не обязательно досконально разбираться в технологии COM, чтобы пользоваться библиотекой DirectX, сведений, приведенных в этом абзаце, вполне достаточно.

Структура приложения Direct3D 10

Если перефразировать одного некогда весьма известного политика, вкратце сущность структуры можно выразить так: "минимальное приложение Direct3D есть минимальное приложение Windows плюс инициализация Direct3D, функция рендеринга (визуализации) и функция очистки памяти". Более наглядно со структурой можно ознакомиться на рис. 2.4.

И действительно, у нас добавятся три новые функции, но прежде, чем мы обратимся к ним, давайте посмотрим, какие изменения нужно внести в функцию `WinMain()`. Исходный текст новой функции `WinMain()` представлен ниже:

```
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow )
{
    // Создаем окно приложения
    if( FAILED( InitWindow( hInstance, nCmdShow ) ) )
           return 0;
    // Инициализируем Direct3D 10
    if( FAILED( InitDirect3D10() ) )
    {
          Cleanup();
          return 0;
```

```
    }

    // Цикл обработки сообщений
    MSG msg = {0};
    while( WM_QUIT != msg.message )
    {
        if( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) )
        {
            TranslateMessage( &msg );
            DispatchMessage( &msg );
        }
        else
        {
            RenderScene();
        }
    }

    Cleanup();
    return (int) msg.wParam;
}
```

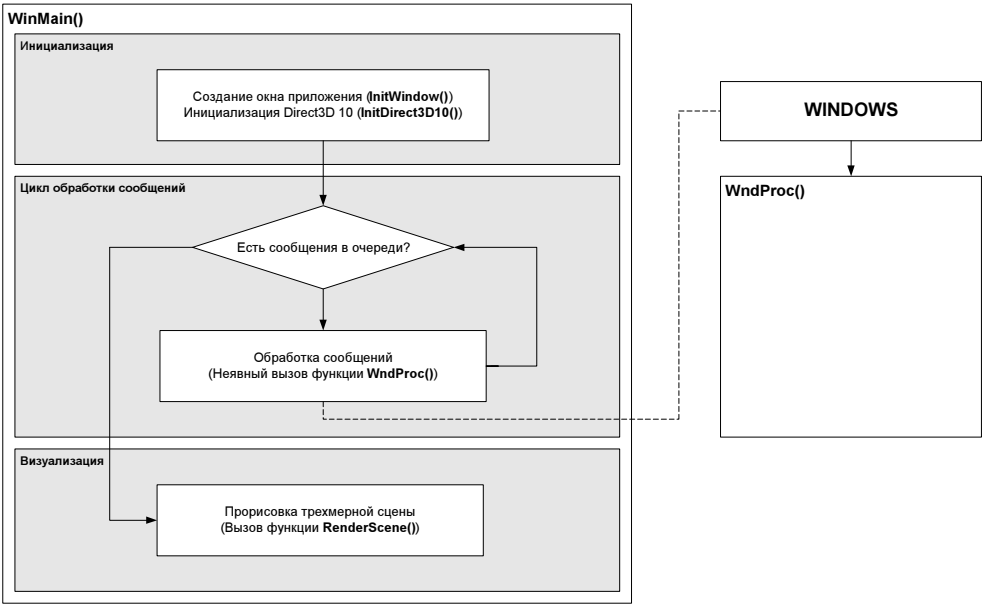

Рис. 2.4. Структура приложения Direct3D

Сразу отметим, что изменилась логика выполнения цикла обработки сообщений. Дело в том, что нас не устраивает работа функции `GetMessage()`. При отсутствии сообщений в очереди цикл из-за нее не продолжается до тех пор, пока какое-либо сообщение не будет поставлено в очередь. Для "обычного" приложения такое положение дел абсолютно нормально, но не для программы, выводящей трехмерную графику в реальном времени. Ведь нам нужно перерисовывать экран несколько раз в секунду!

Функция `PeekMessage()`, в отличие от функции `GetMessage()`, не ждет, пока в очереди появится сообщение, а возвращает ненулевое значение (`TRUE`), если в очереди имеется сообщение для обработки, и нулевое значение (`FALSE`) в противном случае, и выполнение программы после этого продолжается. Прототип функции `PeekMessage()` выглядит так:

```
BOOL PeekMessage(
    LPMSG lpMsg,// Указатель на структуру MSG
    HWND hWnd, // Описатель окна
    UINT wMsgFilterMin, // Нижний предел фильтрации
    UINT wMsgFilterMax, // Верхний предел фильтрации
    UINT wRemoveMsg // Способ обработки сообщений
);
```

Первые четыре параметра функции аналогичны четырем параметрам функции `GetMessage()`. Пятый параметр предписывает, убирать сообщение из очереди после выполнения функции или нет. Мы задаем значение `PM_REMOVE`, указывая на то, что сообщения после выполнения функции необходимо удалять. Таким образом, принимая во внимание особенности функции `PeekMessage()`, проследим логику работы цикла. Сам цикл, как и раньше, выполняется до тех пор, пока не поступит сообщение `WM_QUIT`, с той разницей, что идентификатор поступившего сообщения теперь считывается из переменной `MSG` напрямую. Первой в цикле выполняется функция `PeekMessage()`. Если она возвращает результат `TRUE`, то в очереди есть сообщение, и мы вызываем функции для его обработки (`TranslateMessage()` и `DispatchMessage()`). Если же `PeekMessage()` вернула значение `FALSE`, значит, сообщений для обработки нет. В этом случае у нас есть время выполнить полезную работу — мы вызываем функцию `RenderScene()`, отвечающую за подготовку и вывод графики на экран. Если из очереди извлекается сообщение `WM_QUIT`, цикл завершается, вызывается функция очистки памяти `Cleanup()` и происходит выход из функции `WinMain()`, работа приложения закончена. Думаю, с циклом больше вопросов нет. Переходим к функции инициализации Direct3D 10.

Инициализация Direct3D 10

Прежде всего, давайте в общих чертах познакомимся с алгоритмом вывода графики с помощью Direct3D. Казалось бы, что тут сложного, рисовать и выводить текст на поверхность окна? Для обычных приложений Windows, не выводящих графику в реальном времени, рисование сразу на клиентской области окна вполне годится. Для нас это не подходит. Если рисовать первый кадр, стирать его, и сразу после этого рисовать следующий, то изображение будет мерцать. И дело не в том, что не хватит быстродействия компьютера, а в том, что одни элементы будут оставаться на экране дольше, чем другие. Первый нарисованный элемент мы будем видеть все время, пока рисуется оставшаяся часть кадра, а последний практически сразу будет стерт с экрана — начнется построение следующего кадра. Для устранения этих недостатков вывод графики в Direct3D организован по-другому: изображение рисуется не на экране, а в памяти компьютера, в так называемом *вторичном буфере* (back buffer). Имеется также *первичный буфер*, его содержимое мы и видим на экране. Как только построение во вторичном буфере завершилось, буферы меняются местами, вторичный буфер становится первичным (и отображается на экране), а первичный уходит "в тень", становится вторичным и изображение следующего кадра строится уже в нем. Этот подход называется *двойной буферизацией*. Таким образом, мы получаем практически мгновенную смену изображений. Запомните также, что первичный буфер лишь отображается на экране, вывести на него изображение нельзя, вывод происходит только во вторичный буфер.

Вот теперь мы готовы вплотную подойти к инициализации. Итак, перечислим, какие действия необходимо выполнить для подготовки Direct3D 10 к работе:

- Создать устройство Direct3D 10 (device). Этот объект будет производить рисование кадров в указанный буфер визуализации. Он также оснащен специальными функциями для создания разного рода ресурсов.
- Создать цепочку переключений (swapchain). Цепочка переключений содержит первичный буфер и один или несколько вторичных буферов, отвечает за отображение на экране содержимого буфера, в который устройство выводит изображение, а также за смену кадров.
- Создать объект представления данных как вторичного буфера визуализации (render target view). Представление данных — это аналог приведения типов переменных в языке C (для краткости, дальше так и будем использовать этот термин — "представление данных"). Например, одну и ту же область памяти можно рассматривать как массив данных целого типа, либо как массив данных с плавающей точкой, либо как коды символов

(текст). Байты информации одни и те же, все зависит от того, как их интерпретировать. Буфер в цепочке переключений представляет собой обычную текстуру, она также может содержать данные различного рода. Мы укажем, что данный буфер представляет собой текстуру для рисования. Представление данных позволит нам связать вторичный буфер с устройством, чтобы оно могло выводить в него изображения кадров. Подробнее о текстурах и о ресурсах вообще мы поговорим несколько позднее, в *гл. 4*.

- Установить область отображения (viewport). С помощью области отображения мы можем определить, какую часть вторичного буфера мы будем использовать для вывода графики, говоря более точно — это размеры той области, на которую будет проецироваться наш виртуальный трехмерный мир.

Исходный текст функции `InitDirect3D10()`, которая выполняет все описанные выше действия, может выглядеть примерно так:

```
HRESULT InitDirect3D10()
{
    HRESULT hr = S_OK;

    // Размеры клиентской области окна
    RECT rc;
    GetClientRect( g_hWnd, &rc );
    UINT width = rc.right - rc.left;
    UINT height = rc.bottom - rc.top;

    // Заполняем структуру
    DXGI_SWAP_CHAIN_DESC sd;
    ZeroMemory( &sd, sizeof(sd) );
    sd.BufferCount = 1;
    sd.BufferDesc.Width = width;
    sd.BufferDesc.Height = height;
    sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
    sd.BufferDesc.RefreshRate.Numerator = 60;
    sd.BufferDesc.RefreshRate.Denominator = 1;
    sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
    sd.OutputWindow = g_hWnd;
    sd.SampleDesc.Count = 1;
    sd.SampleDesc.Quality = 0;
    sd.Windowed = TRUE;

    // Пытаемся создать устройство и цепочку переключений
    hr = D3D10CreateDeviceAndSwapChain( NULL,
```

```
            D3D10_DRIVER_TYPE_REFERENCE, NULL, 0, D3D10_SDK_VERSION,
            &sd, &g_pSwapChain, &g_pd3dDevice );
    if( FAILED(hr) )
        return hr;

    // Представление данных как
    // буфера визуализации
    ID3D10Texture2D *pBackBuffer;
    // Получим доступ к вторичному буферу с индексом 0
    hr = g_pSwapChain->GetBuffer( 0, __uuidof( ID3D10Texture2D ),
                                  (LPVOID*)&pBackBuffer );
    if( FAILED(hr) )
        return hr;
    // Создадим представление данных
    hr = g_pd3dDevice->CreateRenderTargetView( pBackBuffer, NULL,
                                               &g_pRenderTargetView );
    pBackBuffer->Release();
    if( FAILED(hr) )
    return hr;

    // Свяжем буфер визуализации с графическим конвейером
    g_pd3dDevice->OMSetRenderTargets( 1, &g_pRenderTargetView, NULL );

    // Настроим область отображения
    D3D10_VIEWPORT vp;
    vp.Width = width;
    vp.Height = height;
    vp.MinDepth = 0.0f;
    vp.MaxDepth = 1.0f;
    vp.TopLeftX = 0;
    vp.TopLeftY = 0;
    g_pd3dDevice->RSSetViewports( 1, &vp );

    return S_OK;
}
```

Функция возвращает результат типа `HRESULT`, он нам уже знаком. Параметров для вызова функции не требуется. В первой строке мы создаем и инициализируем переменную `hr`, она используется в макросах `FAILED` для контроля успешности выполнения функций.

Перед тем как мы создадим устройство и цепочку переключений, подготовим все необходимые данные. Поскольку мы хотим организовать вывод на клиентскую область окна, необходимо выяснить ее размеры. Для этого используем функцию `GetClientRect()`. Функция требует два параметра, первый — описатель окна, размер клиентской области которого нужно узнать, второй — адрес переменной типа `RECT`, в которую функция вернет результат. Вычислим ширину и высоту клиентской части:

```
RECT rc;
GetClientRect( g_hWnd, &rc );
// Ширина и высота
UINT width = rc.right - rc.left;
UINT height = rc.bottom - rc.top;
```

С помощью структурного тип `RECT`, который мы здесь использовали, задаются прямоугольные области: координаты левого верхнего и правого нижнего углов. Объявление типа `RECT` выглядит так:

```
typedef struct RECT {
  LONG left;
  LONG top;
  LONG right;
  LONG bottom;
};
```

Члены структуры несут такую информацию:

- `left` — координата X верхнего левого угла прямоугольника;
- `top` — координата Y верхнего левого угла прямоугольника;
- `right` — координата X нижнего правого угла прямоугольника;
- `bottom` — координата Y нижнего правого угла прямоугольника.

Для создания устройства и цепочки переключений требуется заполнить структуру типа `DXGI_SWAP_CHAIN_DESC`, которая содержит параметры цепочки переключений. Объявим переменную `sd` этого типа и обнулим все значения структуры.

```
DXGI_SWAP_CHAIN_DESC sd;
ZeroMemory( &sd, sizeof(sd) );
```

Зададим количество вторичных буферов, одного нам вполне хватит.

```
sd.BufferCount = 1;
```

Теперь укажем ширину и высоту вторичного буфера в пикселах. У нас размеры вторичного буфера совпадают с размерами клиентской части, а они нам уже известны:

```
sd.BufferDesc.Width = width;
sd.BufferDesc.Height = height;
```

В следующем поле задается формат вторичного буфера. Форматы в Direct3D 10 несколько отличаются от тех, что использовались в девятой версии. Мы установим формат `DXGI_FORMAT_R8G8B8A8_UNORM` (нормированные к интервалу от нуля до единицы 32-битные значения без знака, представляющие собой четыре компонента цвета).

```
sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
```

> ***Замечание***
>
> В Direct3D преимущественно используются форматы, в которых компоненты цвета задаются с помощью чисел с плавающей точкой.

Теперь установим частоту смены кадров, в случае оконного приложения эти значения игнорируются, они имеют смысл только в полноэкранном режиме.

```
sd.BufferDesc.RefreshRate.Numerator = 60;
sd.BufferDesc.RefreshRate.Denominator = 1;
```

Очередное поле определяет использование вторичного буфера. В Direct3D 10 вторичный буфер может использоваться либо как буфер визуализации (`DXGI_USAGE_RENDER_TARGET_OUTPUT`), либо в качестве входных данных для шейдера (`DXGI_USAGE_SHADER_INPUT`). В нашем случае мы запрашиваем его использование как буфера визуализации:

```
sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
```

Определим окно, в которое будет производиться вывод графики. Для этого требуется указать описатель окна:

```
sd.OutputWindow = g_hWnd;
```

Следующие два поля определяют параметры сглаживания изображения с использованием мультисэмплинга, мы его использовать не будем, поэтому впишем следующие значения:

```
sd.SampleDesc.Count = 1;
sd.SampleDesc.Quality = 0;
```

Далее следует параметр, определяющий, в каком режиме будет выводиться графика: в полноэкранном (значение `FALSE`) или в оконном (значение `TRUE`). Установим оконный режим:

```
sd.Windowed = TRUE;
```

Следующее поле определяет, каким образом будет происходить переключение буферов, в том смысле, что происходит с содержимым вторичного буфера после переключения. Мы остановимся на отбрасывании содержимого уже отображенного буфера, так как все равно каждый кадр мы рисуем с нуля:

```
sd.SwapEffect = DXGI_SWAP_EFFECT_DISCARD;
```

Здесь стоит заметить, что значение этого поля можно было бы и не устанавливать. Дело в том, что константе `DXGI_SWAP_EFFECT_DISCARD` соответствует нулевое значение, а после объявления переменной `sd` мы заполнили всю отведенную под нее память нулями, то есть поле `SwapEffect` уже было равно нулю. Тем не менее, забывать о существовании этого поля не следует.

Структуру мы заполнили, теперь вызываем функцию для создания устройства Direct3D 10 и цепочки переключений.

```
hr = D3D10CreateDeviceAndSwapChain( NULL,
            D3D10_DRIVER_TYPE_REFERENCE, NULL, 0, D3D10_SDK_VERSION, &sd,
            &g_pSwapChain, &g_pd3dDevice );
```

Для большей ясности рассмотрим прототип функции:

```
HRESULT D3D10CreateDeviceAndSwapChain(
  IDXGIAdapter *pAdapter,
  D3D10_DRIVER_TYPE DriverType,
  HMODULE Software,
  UINT Flags,
  UINT SDKVersion,
  DXGI_SWAP_CHAIN_DESC *pSwapChainDesc,
  IDXGISwapChain **ppSwapChain,
  ID3D10Device **ppDevice
);
```

Подробно разберем параметры:

- `pAdapter` — указатель на интерфейс `IDXGIAdapter`, который определяет, к какому видеоадаптеру в системе будет привязано создаваемое устройство, значение `NULL` означает использование видеоадаптера по умолчанию;
- `DriverType` — тип драйвера создаваемого устройства, если данный видеоадаптер поддерживает DirectX 10, можно создать устройство, использующее аппаратное ускорение графики, указав тип `D3D10_DRIVER_TYPE_HARDWARE`, в противном случае можно создать устройство, использующее программную эмуляцию функций Direct3D 10, задав тип `D3D10_DRIVER_TYPE_REFERENCE`;
- `Software` — описатель библиотеки DLL, в которой реализован программный растеризатор, используется только в случае, если указан тип драйвера `D3D10_DRIVER_TYPE_SOFTWARE` (этот тип зарезервирован для использования в будущих версиях), иначе должно указываться значение `NULL`;
- `Flags` — необязательные флаги создания устройства, которые регулируют использование дополнительных возможностей API, например, для вывода дополнительной отладочной информации и расширения возможностей отладки можно указать флаг `D3D10_CREATE_DEVICE_DEBUG`;

- `SDKVersion` — здесь всегда используется значение `D3D10_SDK_VERSION`, это номер версии SDK, он определен в файле d3d10.h;
- `pSwapChainDesc` — указатель на переменную типа `DXGI_SWAP_CHAIN_DESC`, в ней, как мы уже знаем, содержатся параметры создаваемой цепочки переключений;
- `ppSwapChain` — адрес указателя на интерфейс `IDXGISwapChain`, этот указатель функция установит на вновь созданную цепочку переключений;
- `ppDevice` — адрес указателя на интерфейс `ID3D10Device`, этот указатель функция установит на вновь созданное устройство.

Теперь, зная, что означают параметры, можно перевести вызов функции на понятный для нас язык: создать устройство Direct3D 10 и цепочку переключений. Использовать видеоадаптер по умолчанию (первый обнаруженный), тип драйвера устройства — программная эмуляция, параметры цепочки переключений взять из переменной `sd`, указатели на интерфейсы цепочки переключений и созданного устройства поместить в переменные `g_pSwapChain` и `g_pd3dDevice`, соответственно. Скорее всего, к моменту выхода книги в печать совместимые с DirectX 10 видеокарты будут уже доступны по разумным ценам. При наличии такой видеокарты и соответствующих драйверов, можно спокойно создавать устройство с драйвером типа `D3D10_DRIVER_TYPE_HARDWARE`, тем не менее, пока у большинства из нас таких видеокарт нет, будем довольствоваться программной эмуляцией.

Устройство и цепочку переключений мы создали. Теперь нужно связать устройство с вторичным буфером цепочки переключений. Объявим временную переменную — указатель на текстуру.

```
ID3D10Texture2D *pBackBuffer;
```

После этого получим доступ к вторичному буферу и поместим его адрес в только что объявленную переменную. Для этого вызовем метод `GetBuffer()` цепочки переключений (`g_pSwapChain`):

```
hr = g_pSwapChain->GetBuffer( 0, __uuidof( ID3D10Texture2D ),
                              (LPVOID*)&pBackBuffer );
```

Рассмотрим прототип этого метода.

```
HRESULT GetBuffer(
  UINT Buffer,
  REFIID riid,
  void **ppSurface
);
```

Параметры имеют следующий смысл:

- `Buffer` — индекс вторичного буфера, к которому нужно получить доступ;

- `riid` — идентификатор интерфейса (GUID), который используется для выполнения действий с буфером;
- `ppSurface` — адрес указателя на интерфейс вторичного буфера, этот указатель метод установит на вторичный буфер.

Необходимо прокомментировать использование макроса со странным именем `__uuidof`, он возвращает GUID, закрепленный за аргументом. Результатом вызова метода `GetBuffer()` будет указатель на вторичный буфер. Причем указатель этот будет на интерфейс `ID3D10Texture2D`, то есть на двухмерную текстуру. Имея такой указатель, мы создадим представление данных как вторичного буфера визуализации. Это делается вызовом метода `CreateRenderTargetView` устройства Direct3D 10:

```
hr = g_pd3dDevice->CreateRenderTargetView( pBackBuffer, NULL,
        &g_pRenderTargetView );
```

Прототип метода выглядит следующим образом:

```
HRESULT CreateRenderTargetView(
  ID3D10Resource *pResource,
  const D3D10_RENDER_TARGET_VIEW_DESC *pDesc,
  ID3D10RenderTargetView **ppRTView
);
```

Опишем смысл параметров:

- `pResource` — указатель на ресурс, содержащий данные;
- `pDesc` — указатель на структуру типа `D3D10_RENDER_TARGET_VIEW_DESC`, содержащую описание параметров буфера визуализации (подстановка значения `NULL` означает, что мы используем параметры по умолчанию);
- `ppRTView` — адрес указателя на созданное представление данных, который будет установлен методом.

Таким образом, мы просим создать представление данных для вывода информации о пикселах кадра, используя указатель на текстуру `pBackBuffer`, а указатель на созданное представление данных поместить в переменную `g_pRenderTargetView`. После выполнения метода мы освобождаем указатель `pBackBuffer`, т. к. он нам больше не потребуется:

```
pBackBuffer->Release();
```

Полученное представление данных мы используем для того, чтобы назначить устройству Direct3D 10 буфер, в который оно будет выводить изображение. Здесь нам пригодится метод `OMSetRenderTargets`:

```
g_pd3dDevice->OMSetRenderTargets( 1, &g_pRenderTargetView, NULL );
```

Рассмотрим прототип метода и разберем его параметры.

```
void OMSetRenderTargets(
  UINT NumViews,
  ID3D10RenderTargetView *const *ppRenderTargetViews,
  ID3D10DepthStencilView *pDepthStencilView
);
```

- `NumViews` — количество представлений данных для связывания с устройством;
- `ppRenderTargetViews` — указатель на массив представлений данных для буферов визуализации, которые нужно связать с устройством;
- `pDepthStencilView` — указатель на представление данных как шаблонного буфера глубины.

Итак, вызывая этот метод, мы просим связать единственное представление данных (`g_pRenderTargetView`) с устройством, а представление для шаблонного буфера глубины не использовать (с шаблонным буфером глубины мы познакомимся в *гл. 8*).

Ну вот, мы наконец-то объяснили устройству, куда выводить данные. Осталось задать область отображения и можно выводить графику! Область отображения устанавливается с помощью метода `RSSetViewports` устройства Direct3D 10. Однако прежде чем его вызвать, необходимо заполнить структуру типа `D3D10_VIEWPORT`, содержащую параметры области отображения. Объявим переменную `vp`:

```
D3D10_VIEWPORT vp;
```

Укажем ширину и высоту области отображения, используя рассчитанные размеры клиентской части окна приложения:

```
vp.Width = width;
vp.Height = height;
```

Зададим минимальную и максимальную глубину, которые будет охватывать область отображения:

```
vp.MinDepth = 0.0f;
vp.MaxDepth = 1.0f;
```

В следующих двух полях задается смещение области отображения от левой и верхней границ текстуры, на которую производится визуализация. Поскольку мы используем для вывода всю поверхность текстуры, смещения у нас нет:

```
vp.TopLeftX = 0;
vp.TopLeftY = 0;
```

Теперь, когда все данные для установки области отображения у нас имеются, вызываем метод `RSSetViewports`:

```
g_pd3dDevice->RSSetViewports( 1, &vp );
```

Вот его прототип:

```
void RSSetViewports(
  UINT NumViewports,
  const D3D10_VIEWPORT *pViewports
);
```

Параметров всего два:

- `NumViewports` — количество областей отображения, которые необходимо установить;
- `pViewports` — указатель на массив структур, описывающих области отображения, которые необходимо связать с устройством.

Как следует из нашего вызова, мы связываем с устройством одну область отображения, указатель на описание которой находится в переменной `vp`.

Все, дело сделано, остается только выйти из функции инициализации, вернув результат успешного завершения:

```
return S_OK;
```

Ну вот, самое сложное теперь позади. Перейдем к функции визуализации (рисования) трехмерной сцены `RenderScene()`. Напомню, приложение вызывает эту функцию, когда в очереди нет сообщений. Выглядеть функция может примерно следующим образом:

```
void RenderScene()
{
  // Очищаем вторичный буфер
  // (компоненты красного, зеленого, синего, прозрачность)
  float ClearColor[4] = { 0.1f, 0.5f, 0.1f, 1.0f };
  g_pd3dDevice->ClearRenderTargetView( g_pRenderTargetView, ClearColor );
  g_pSwapChain->Present( 0, 0 );
}
```

У нас, конечно, пока практически ничего на экран не выводится. Наша функция только закрашивает вторичный буфер указанным нами цветом и вызывает переключение буферов.

В первой строке устанавливается цвет, которым будет закрашен вторичный буфер:

```
float ClearColor[4] = { 0.1f, 0.5f, 0.1f, 1.0f };
```

Здесь мы задаем компоненты цвета и прозрачность, из указанных значений получится темноватый зеленый цвет, полностью непрозрачный.

Для закрашивания вторичного буфера используется метод `ClearRenderTargetView` устройства Direct3D 10:

```
g_pd3dDevice->ClearRenderTargetView( g_pRenderTargetView, ClearColor );
```

Прототип метода выглядит следующим образом:

```
void ClearRenderTargetView(
  ID3D10RenderTargetView *pRenderTargetView,
  const FLOAT ColorRGBA[4]
);
```

где

- `pRenderTargetView` — указатель на представление данных вторичного буфера визуализации;
- `ColorRGBA[4]` — массив, содержащий компоненты цвета (RGBA).

После того как завершено построение сцены, в нашем случае — когда закрашен вторичный буфер, мы должны переключить буферы. Вторичный буфер станет первичным, и мы сможем увидеть то, что на нем нарисовано. Первичный, в свою очередь, станет вторичным, рисование следующего кадра будет производиться уже на нем, и т. д. Используем метод `Present()` цепочки переключений:

```
g_pSwapChain->Present( 0, 0 );
```

Рассмотрим прототип метода и его параметры:

```
HRESULT Present(
  UINT SyncInterval,
  UINT Flags
);
```

- `SyncInterval` — определяет интервал синхронизации изображения, его возможные значения:
  - `0` — без синхронизации, вывод происходит немедленно;
  - `1..4` — синхронизация с n-м обратным ходом кадровой развертки монитора;
- `Flags` — флаги, определяющие режим работы метода:
  - `0` — обычный вывод на экран;
  - `DXGI_PRESENT_TEST` — тест состояния цепочки переключений, без вывода изображения (используется для получения информации о том, не заслонено ли окно программы каким-либо другим окном).

В нашем случае происходит немедленный вывод изображения без синхронизации с кадровой разверткой. Рисование кадров и переключение буферов будет производиться до тех пор, пока не придет время завершать работу приложения (то есть поступит сообщение `WM_QUIT`). Перед выходом из программы мы должны освободить все задействованные в нашей программе ресурсы (очистить память). В нашем приложении за это отвечает функция `Cleanup()`, вот ее исходный текст:

```
void Cleanup()
{
  if( g_pd3dDevice ) g_pd3dDevice->ClearState();

  if( g_pRenderTargetView ) g_pRenderTargetView->Release();
  if( g_pSwapChain ) g_pSwapChain->Release();
  if( g_pd3dDevice ) g_pd3dDevice->Release();
}
```

В каждой строке с помощью оператора `if` производится проверка на наличие объекта в памяти (в это случае указатель не равен `NULL`), только после этого можно вызывать какие-то его методы. В начальной части функции мы вызываем метод `ClearState()` устройства Direct3D 10, он возвращает все настройки устройства в состояние по умолчанию, какими они были сразу после его создания:

```
if( g_pd3dDevice ) g_pd3dDevice->ClearState();
```

Оставшиеся три строки вызывают методы `Release` соответствующих объектов, с помощью которых происходит освобождение памяти:

```
if( g_pRenderTargetView ) g_pRenderTargetView->Release();
if( g_pSwapChain ) g_pSwapChain->Release();
if( g_pd3dDevice ) g_pd3dDevice->Release();
```

Результат работы минимального приложения Direct3D 10 можно увидеть на рис. 2.5.

Полный исходный текст приложения приведен в листинге 2.2, соответствующий проект можно найти на прилагаемом компакт-диске в директории Glava2\Min_Direct3D10.

Прокомментируем некоторые моменты, которых мы еще не касались. В самом начале, рядом с подключением файла windows.h, необходимо также подключить заголовочные файлы Direct3D 10: d3d10.h и d3dx10.h. Не забываем, что кроме этого нужно подключить библиотеки d3d10.lib и d3dx10d.lib в свойствах проекта, как мы уже говорили, их нужно добавлять во все проекты, где используется Direct3D 10.

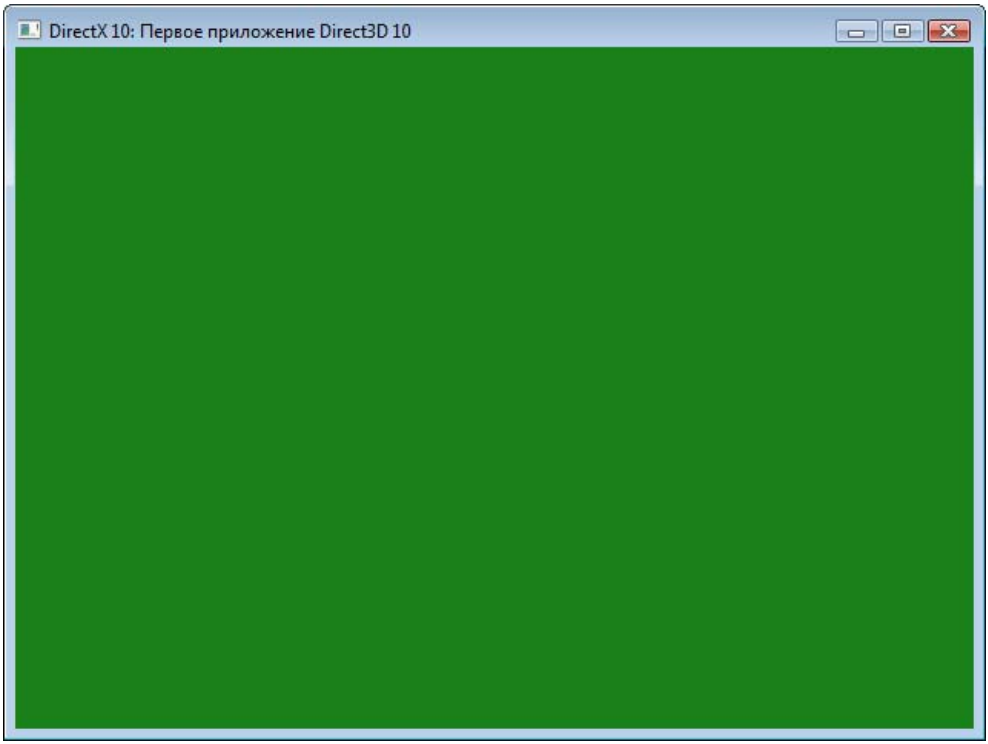

Рис. 2.5. Минимальное приложение Direct3D 10

К глобальным переменным добавилась переменная, хранящая тип драйвера (`g_driverType`), а также указатели на интерфейсы устройства (`g_pd3dDevice`), на цепочку переключения (`g_pSwapChain`) и на интерфейс представления данных буфера визуализации (`g_pRenderTargetView`). В прототипы функций введены прототипы для всех новых функций: `InitDirect3D10()`, `RenderScene()` и `Cleanup()`. Функция `InitDirect3D10()` несколько отличается от рассмотренной нами выше. Перед заполнением структуры `sd` мы объявляем массив, содержащий типы драйверов устройства:

```
D3D10_DRIVER_TYPE driverTypes[] =
{
   D3D10_DRIVER_TYPE_HARDWARE,
   D3D10_DRIVER_TYPE_REFERENCE,
};
```

Вычисляем количество элементов, содержащееся в массиве:

```
UINT numDriverTypes = sizeof(driverTypes) / sizeof(driverTypes[0]);
```

После заполнения структуры `sd` запускается цикл, в котором из массива по очереди извлекается элемент с очередным типом драйвера устройства и производится попытка создания устройства Direct3D 10 с использованием драйвера данного типа:

```
for( UINT driverTypeIndex = 0; driverTypeIndex < numDriverTypes;
        driverTypeIndex++ )
       {
              g_driverType = driverTypes[driverTypeIndex];
              hr = D3D10CreateDeviceAndSwapChain( NULL, g_driverType,
                           NULL, 0, D3D10_SDK_VERSION, &sd,
                           &g_pSwapChain, &g_pd3dDevice );
              if( SUCCEEDED( hr ) )
                     break;
       }
```

При первой удачной попытке цикл прерывается, и дальше функция ничем не отличается от знакомого нам варианта. Предвижу вопрос: зачем нужно такое усложнение? Это сделано для того, чтобы программу можно было без перекомпиляции запускать как на компьютерах с новыми видеокартами, совместимыми с DirectX 10, так и на других машинах, где Direct3D 10 еще аппаратно не поддерживается.

Листинг 2.2

```
//--------------------------------------------------------------------------
// Минимальное приложение Direct3D 10
//--------------------------------------------------------------------------
#include <windows.h>
#include <d3d10.h>
#include <d3dx10.h>

// Ширина и высота окна
#define WINDOW_WIDTH  640
#define WINDOW_HEIGHT 480

//--------------------------------------------------------------------------
// Глобальные переменные
//--------------------------------------------------------------------------
HWND                    g_hWnd = NULL;
D3D10_DRIVER_TYPE       g_driverType = D3D10_DRIVER_TYPE_NULL;
```

```
ID3D10Device*           g_pd3dDevice = NULL;
IDXGISwapChain*         g_pSwapChain = NULL;
ID3D10RenderTargetView* g_pRenderTargetView = NULL;

//--------------------------------------------------------------------------
// Прототипы функций
//--------------------------------------------------------------------------
HRESULT             InitWindow( HINSTANCE hInstance, int nCmdShow );
HRESULT             InitDirect3D10();
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);
void                Cleanup();
void                RenderScene();

//--------------------------------------------------------------------------
// С этой функции начинается выполнение программы
//--------------------------------------------------------------------------
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow )
{
  // Создаем окно приложения
  if( FAILED( InitWindow( hInstance, nCmdShow ) ) )
    return 0;
       //
       if( FAILED( InitDirect3D10() ) )
       {
              Cleanup();
              return 0;
       }

       // Цикл обработки сообщений
       MSG msg = {0};
       while( WM_QUIT != msg.message )
       {
              if( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) )
              {
                     TranslateMessage( &msg );
                     DispatchMessage( &msg );
              }
              else
```

```
            {
                RenderScene();
            }

        }
        return (int) msg.wParam;
}

//--------------------------------------------------------------------------
// Регистрация класса и создание окна
//--------------------------------------------------------------------------
HRESULT InitWindow( HINSTANCE hInstance, int nCmdShow )
{
        // Регистрируем класс окна
        WNDCLASSEX wc;
        wc.cbSize         = sizeof(WNDCLASSEX);
        wc.style          = CS_HREDRAW | CS_VREDRAW;
        wc.lpfnWndProc    = WndProc;
        wc.cbClsExtra     = 0;
        wc.cbWndExtra     = 0;
        wc.hInstance      = hInstance;
        wc.hIcon          = NULL;
        wc.hCursor        = LoadCursor(NULL, IDC_ARROW);
        wc.hbrBackground  = (HBRUSH)COLOR_WINDOW;
        wc.lpszMenuName   = NULL;
        wc.lpszClassName  = L"SimpleWindowClass";
        wc.hIconSm        = LoadIcon(NULL, IDI_APPLICATION);
        if( !RegisterClassEx(&wc) )
            return E_FAIL;

        // Создаем окно
        g_hWnd = CreateWindow(
            L"SimpleWindowClass",
            L"DirectX 10: Первое приложение Direct3D 10",
            WS_OVERLAPPEDWINDOW,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            WINDOW_WIDTH,
            WINDOW_HEIGHT,
```

```
            NULL,
            NULL,
            hInstance,
            NULL);

    // Если не удалось создать окно - выходим из функции
    if( !g_hWnd )
            return E_FAIL;
    // Отображаем окно на экране
    ShowWindow( g_hWnd, nCmdShow );
    UpdateWindow(g_hWnd);

    return S_OK;
}

//--------------------------------------------------------------------------
// Инициализация Direct3D
//--------------------------------------------------------------------------
HRESULT InitDirect3D10()
{
    HRESULT hr = S_OK;

    // Размеры клиентской области окна
    RECT rc;
    GetClientRect( g_hWnd, &rc );
    UINT width = rc.right - rc.left;
    UINT height = rc.bottom - rc.top;

    // Список возможных типов устройства
    D3D10_DRIVER_TYPE driverTypes[] =
    {
            D3D10_DRIVER_TYPE_HARDWARE,
            D3D10_DRIVER_TYPE_REFERENCE,
    };
    UINT numDriverTypes =
                    sizeof(driverTypes) / sizeof(driverTypes[0]);

    // Заполняем структуру
    DXGI_SWAP_CHAIN_DESC sd;
```

```
        ZeroMemory( &sd, sizeof(sd) );
        sd.BufferCount = 1;
        sd.BufferDesc.Width = width;
        sd.BufferDesc.Height = height;
        sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
        sd.BufferDesc.RefreshRate.Numerator = 60;
        sd.BufferDesc.RefreshRate.Denominator = 1;
        sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
        sd.OutputWindow = g_hWnd;
        sd.SampleDesc.Count = 1;
        sd.SampleDesc.Quality = 0;
        sd.Windowed = TRUE;

        // Пытаемся создать устройство и цепочку переключений,
        // проходя по списку
  // как только получилось - выходим из цикла
  for( UINT driverTypeIndex = 0; driverTypeIndex < numDriverTypes;
               driverTypeIndex++ )
        {
               g_driverType = driverTypes[driverTypeIndex];
               hr = D3D10CreateDeviceAndSwapChain( NULL, g_driverType,
                              NULL, 0, D3D10_SDK_VERSION, &sd,
                              &g_pSwapChain, &g_pd3dDevice );
               if( SUCCEEDED( hr ) )
                      break;
        }
        if( FAILED(hr) )
               return hr;
// Представление данных для буфера визуализации
  ID3D10Texture2D *pBackBuffer;
// Получим доступ к вторичному буферу с индексом 0
  hr = g_pSwapChain->GetBuffer( 0, __uuidof( ID3D10Texture2D ),
         (LPVOID*)&pBackBuffer );
  if( FAILED(hr) )
    return hr;
// Создадим представление данных
  hr = g_pd3dDevice->CreateRenderTargetView( pBackBuffer, NULL,
                          &g_pRenderTargetView );
  pBackBuffer->Release();
```

```
  if( FAILED(hr) )
    return hr;

// Свяжем буфер визуализации с графическим конвейером
  g_pd3dDevice->OMSetRenderTargets( 1, &g_pRenderTargetView, NULL );

  // Настроим область отображения
  D3D10_VIEWPORT vp;
  vp.Width = width;
  vp.Height = height;
  vp.MinDepth = 0.0f;
  vp.MaxDepth = 1.0f;
  vp.TopLeftX = 0;
  vp.TopLeftY = 0;
  g_pd3dDevice->RSSetViewports( 1, &vp );

  return S_OK;
}

//--------------------------------------------------------------------------
// Прорисовка трехмерной сцены
//--------------------------------------------------------------------------
void RenderScene()
{
       // Очищаем вторичный буфер
       //(компоненты красного, зеленого, синего, прозрачность)
       float ClearColor[4] = { 0.1f, 0.5f, 0.1f, 1.0f };
       g_pd3dDevice->ClearRenderTargetView( g_pRenderTargetView,
                          ClearColor );
       g_pSwapChain->Present( 0, 0 );
}
//--------------------------------------------------------------------------
// Обработка сообщений
//--------------------------------------------------------------------------
LRESULT CALLBACK WndProc( HWND hWnd, UINT message, WPARAM wParam,
                          LPARAM lParam )
{
  switch (message)
```

```
  {

        case WM_DESTROY:
              PostQuitMessage(0);
              break;
        default:
              return DefWindowProc(hWnd, message, wParam, lParam);
        }

        return 0;
}

//--------------------------------------------------------------------------
// Очищаем память
//--------------------------------------------------------------------------
void Cleanup()
{
  if( g_pd3dDevice ) g_pd3dDevice->ClearState();

  if( g_pRenderTargetView ) g_pRenderTargetView->Release();
  if( g_pSwapChain ) g_pSwapChain->Release();
  if( g_pd3dDevice ) g_pd3dDevice->Release();
}
```

Особенности полноэкранного режима

Мы рассмотрели создание оконного приложения, которое использует возможности Direct3D 10. Для полноты картины нужно рассмотреть и работу в полноэкранном режиме. Для этого давайте сначала познакомимся с DXGI.

Вы наверняка заметили, что указатель на интерфейс устройства и на цепочку переключений имеют разные типы (`ID3D10Device` и `IDXGISwapChain`, соответственно). DXGI (DirectX Graphics Infrastructure) — это графическая инфраструктура DirectX, ее задача состоит в управлении низкоуровневыми операциями. Она обеспечивает единую основу для графических компонентов. Direct3D 10 — первый из компонентов, который использует преимущества DXGI. В отличие от предыдущих версий DirectX, такие операции низкого уровня, как перечисление устройств в системе, управление яркостью и контрастностью изображения, переключение в полноэкранный режим, сейчас полностью реализованы с помощью DXGI, которая общается с драйвером привилегированного режима и видеоадаптером системы.

Есть два способа взаимодействия с DXGI: напрямую и через посредничество API Direct3D 10. Взаимодействие напрямую необходимо, если требуется выполнить какое-либо низкоуровневые операции, например, перечисление всех видеоадаптеров в системе, или, как мы сейчас увидим, для переключения в полноэкранный режим и обратно.

Вообще говоря, чтобы создать приложение, работающее в полноэкранном режиме, достаточно при заполнении структуры `sd` указать:

```
sd.Windowed = FALSE;
```

Для нормальной работы такого приложения потребуется внести еще кое-какие изменения. Под нормальной работой понимаем возможность выхода из программы, переключение из полноэкранного режима в оконный и обратно.

В предыдущих версиях DirectX работа в полноэкранном режиме была сопряжена с некоторыми сложностями, так как существовала возможность "потерять" устройство Direct3D. "Потеря" происходила, например, если при работе приложения в полноэкранном режиме пользователь нажимал комбинацию клавиш <Alt>+<Tab>, переключаясь на другое приложение, либо если запускалось еще одно приложение DirectX, также требовавшее для работы полноэкранного режима. "Потерянное" устройство не могло выводить информацию на экран, и при этом не выдавало никакого сообщения об ошибке. Для возвращения в полноэкранный режим приходилось сбрасывать устройство и пересоздавать все созданные ресурсы. В DirectX 10 жизнь стала немного проще: "потери устройства" не происходит, происходит "потеря полноэкранного режима", приложение просто переходит в оконный режим. Чтобы вернуться в полноэкранный режим, нужно либо нажать стандартную комбинацию клавиш — <Alt>+<Enter>, либо специальную клавишу переключения режимов, предусмотренную в приложении.

Итак, рассмотрим, какие шаги нужно предпринять для написания программы, которая может переключаться в полноэкранный режим и обратно по нажатию <Alt>+<Enter> либо клавишей <F1>. Переключение с помощью комбинаций клавиш <Alt>+<Enter> реализована по умолчанию. Таким образом, приложение переводится в полноэкранный режим, где используется текущее разрешение экрана. Если необходимо запретить использование этой комбинации клавиш, необходимо вызвать метод `IDXGIFactory::MakeWindowAssociation` с соответствующими параметрами. Выглядеть это может, например, так: сначала мы в функции `InitDirect3D10()` создаем указатель на интерфейс `IDXGIFactory`:

```
hr = CreateDXGIFactory(__uuidof(IDXGIFactory), (void**)(&g_pFactory) );
```

А потом в нужном месте программы вызываем необходимый метод:

```
g_pFactory->MakeWindowAssociation(g_hWnd, DXGI_MWA_NO_ALT_ENTER);
```

Для корректного освобождения памяти в функции `Cleanup()` нужно добавить строку

```
if (g_pFactory) g_pFactory->Release();
```

Но это мы отвлеклись. Нас интересует, как реализуется режим переключения, а не как он отключается. Для программного переключения в полноэкранный режим и обратно нужно вызвать метод `IDXGISwapChain::SetFullscreenState`. Прототип метода выглядит следующим образом:

```
HRESULT SetFullscreenState(
  BOOL Fullscreen,
  IDXGIOutput *pTarget
);
```

Параметры метода имеют следующий смысл.

- `Fullscreen` — значение `TRUE` указывается для переключения в полноэкранный режим, `FALSE` — для переключения в оконный.
- `pTarget` — указатель на интерфейс `IDXGIOutput`, отвечающий за вывод изображения на монитор. Грубо говоря, он определяет, на каком из подключенных мониторов переключить изображение на полный экран. (Поскольку мы рассматриваем вывод изображения на один монитор, здесь мы будем использовать значение `NULL` для его автоматического определения.)

В программе удобно закрепить переключение режимов за какой-то клавишей, мы договорились использовать <F1>. Добавим обработку события `WM_KEYDOWN` (оно сигнализирует о нажатии клавиши на клавиатуре) в оконную процедуру:

```
case WM_KEYDOWN:
    if (VK_F1 == wParam)
    {
        BOOL InFullScreen;
        g_pSwapChain->GetFullscreenState(&InFullScreen, NULL);
        g_pSwapChain->SetFullscreenState(!InFullScreen, NULL);
    }
break;
```

Логика работы здесь следующая. При поступлении события `WM_KEYDOWN` и при условии, что нажата клавиша <F1>, создается локальная переменная `InFullScreen`, и в нее с помощью метода `GetFullscreenState()` заносится текущий режим вывода изображения. Этот метод является методом `SetFullscreenState()` "наоборот": он извлекает режим вывода на экран в указанную переменную. После его вызова происходит переключение режима на обратный тому, который установлен в настоящее время. Если был

оконный — будет полноэкранный, и наоборот. С переключением такой метод справится очень хорошо, но результат нас, скорее всего, не устроит. Мы получим все то же окно 640×480, только растянутое во весь экран. Если вы когда-нибудь увеличивали маленькую картинку на весь экран, вы понимаете, о чем речь. Для получения нормального результата необходимо при переключении производить соответствующее изменение размеров вторичных буферов и буфера визуализации. Метод `IDXGISwapChain::SetFullscreenState` при переключении режимов посылает приложению сообщение `WM_SIZE`, которое возникает также при изменении размеров окна. Все изменения размеров буферов мы будем производить именно в обработчике этого события. Удобно выделить все эти операции в отдельную функцию, назовем ее `ResizeBuffers()`. Обработчик события будет выглядеть следующим образом:

```
case WM_SIZE:
    if (g_InitDone) ResizeBuffers();
break;
```

Здесь есть небольшая тонкость: сообщение `WM_SIZE` приходит также и от операционной системы, и если программа у нас запускается сразу в полноэкранном режиме, то это сообщение придет еще до завершения функции инициализации. Это, конечно, приведет к вызову функции `ResizeBuffers()`, которая завершится с ошибкой, потому что будет пытаться работать с еще не созданными объектами. Для устранения этой проблемы мы введем глобальную переменную `g_InitDone`. При объявлении этой переменной будет присвоено значение `FALSE`, а значение `TRUE` она получит только перед выходом из функции `InitDirect3D10()`. Таким образом, вызов функции будет производиться гарантированно после инициализации, и программа будет работать корректно. Рассмотрим теперь саму функцию `ResizeBuffers()`. Вот минимальный объем действий, который она должна выполнять:

1. Освободить все указатели на вторичные буферы и представления данных для буфера визуализации.
2. Выполнить изменение размеров вторичных буферов.
3. Создать представление данных для буфера визуализации и связать его с графическим конвейером (аналогично тому, как мы это делали при создании устройства Direct3D 10).
4. Скорректировать размеры области отображения.

Рассмотрим действия функции по порядку. Начнем с объявления функции и переменной для контроля наличия ошибок:

```
HRESULT ResizeBuffers()
{
  HRESULT hr = S_OK;
```

Освобождаем ссылку на представление данных буфера визуализации:

```
if( g_pRenderTargetView ) g_pRenderTargetView->Release();
```

Вызываем соответствующий метод для изменения размеров буферов:

```
if ( g_pSwapChain )
 g_pSwapChain->ResizeBuffers(2, 0, 0, DXGI_FORMAT_R8G8B8A8_UNORM, 0)
```

Познакомимся с прототипом этого метода:

```
HRESULT ResizeBuffers(
  UINT BufferCount,
  UINT Width,
  UINT Height,
  DXGI_FORMAT NewFormat,
  UINT SwapChainFlags
);
```

Вот описание его параметров:

- `BufferCount` — количество буферов в цепочке переключений (общее количество, включающее первичный буфер и все вторичные);
- `Width` — новая ширина вторичного буфера, если указано значение 0, автоматически принимается равным ширине клиентской области окна, куда производится вывод;
- `Height` — новая высота вторичного буфера, аналогично, если указано значение 0, автоматически принимается равным высоте клиентской области окна, куда производится вывод;
- `NewFormat` — новый формат вторичного буфера (мы указываем тот же самый, что при создании устройства Direct3D 10);
- `SwapChainFlags` — дополнительные флаги, задающие параметры переключения буферов, здесь мы их не используем.

После изменения размеров вторичных буферов мы практически полностью повторяем действия в функции `InitDirect3D10()` после создания устройства:

```
ID3D10Texture2D *pBackBuffer;
  hr = g_pSwapChain->GetBuffer( 0, __uuidof( ID3D10Texture2D ),
            (LPVOID*)&pBackBuffer );
  if( FAILED(hr) )
    return hr;

  hr = g_pd3dDevice->CreateRenderTargetView( pBackBuffer, NULL,
            &g_pRenderTargetView );
```

```
  pBackBuffer->Release();
  if( FAILED(hr) )
    return hr;

  g_pd3dDevice->OMSetRenderTargets( 1, &g_pRenderTargetView, NULL );
```

Далее мы должны скорректировать область отображения. Для этого нам нужно узнать размеры клиентской области окна, либо разрешение экрана, если переключение произошло в полноэкранный режим:

```
  BOOL InFullScreen;
  UINT width = 0;
  UINT height = 0;
// Определим, какой режим сейчас установлен
  g_pSwapChain->GetFullscreenState( &InFullScreen, NULL );
  if ( !InFullScreen )
  {
    // Оконный режим:
    // определяем размеры клиентской области окна
    RECT rc;
    GetClientRect( g_hWnd, &rc );
    width = rc.right - rc.left;
    height = rc.bottom - rc.top;
  } else
  {
    // Полноэкранный режим:
    // определяем разрешение экрана
    width = GetSystemMetrics (SM_CXSCREEN);
    height = GetSystemMetrics (SM_CYSCREEN);
  }

  // Настроим область отображения
  D3D10_VIEWPORT vp;
  vp.Width = width;
  vp.Height = height;
  vp.MinDepth = 0.0f;
  vp.MaxDepth = 1.0f;
  vp.TopLeftX = 0;
  vp.TopLeftY = 0;
```

```
  g_pd3dDevice->RSSetViewports( 1, &vp );

  return S_OK;
}
```

Логика, наверное, понятна и без пояснений: определяем, в каком режиме работает программа, и в соответствии с этим вычисляем новые размеры ширины и высоты. После заполнения структуры и установки новых размеров области отображения работа функции завершается.

Вот так осуществляется переключение между оконным и полноэкранным режимами. Конкретный пример приводить здесь мы пока не будем, с полноэкранным режимом мы еще столкнемся в *гл. 6*. А пока что нам есть чем заняться: для программирования трехмерной графики нам потребуются знания из некоторых разделов математики, посмотрим, что там к чему.

Глава 3

Немного математики

Системы координат в DirectX 10

Что представляет собой система координат? Это система отсчета, с помощью которой положение объекта можно описать расстоянием от "нулевой точки", точки начала координат. Простой пример одномерной системы координат — это прямая дорога с километровыми столбиками. В этой координатной системе началом отсчета, "нулевой точкой", является город, из которого дорога выходит. Таким образом, выехав из города, на вопрос по мобильному телефону, "где ты сейчас находишься?", можно ответить: "только что проехал девятнадцатый километр", и ответ будет исчерпывающий. На рис. 3.1 можно увидеть эту самую дорогу.

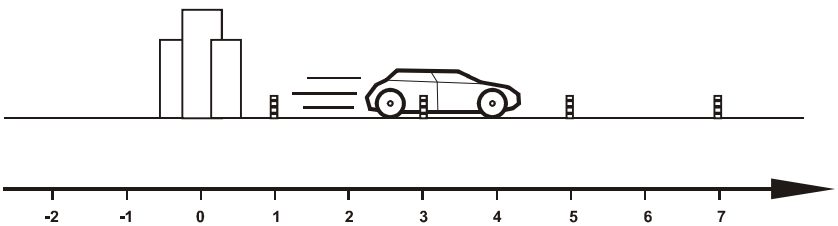

Рис. 3.1. Пример одномерной системы координат

С двухмерной системой координат также все просто. Представим зал кинотеатра. В билете указываются координаты места: номер ряда и номер кресла в этом ряду (рис. 3.2). Как можно заметить, двухмерная система координат представляет собой две одномерные системы, соединенные под прямым углом и имеющие общее начало координат. Горизонтальная координатная ось называется осью X, вертикальная — осью Y. Положение точки задается двумя значениями: расстояниями от начала координат по оси X и по оси Y.

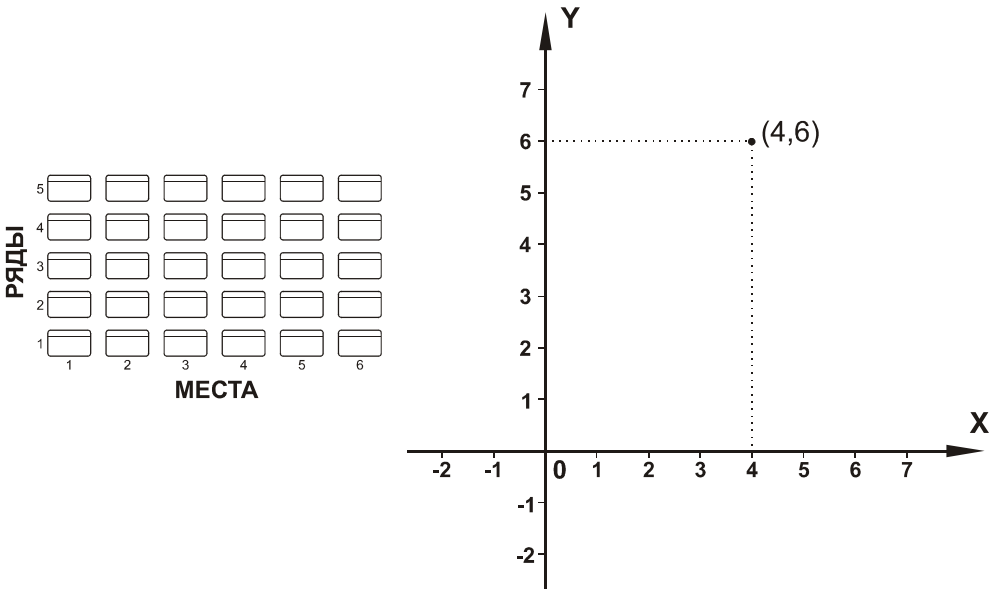

Рис. 3.2. Двухмерная система координат

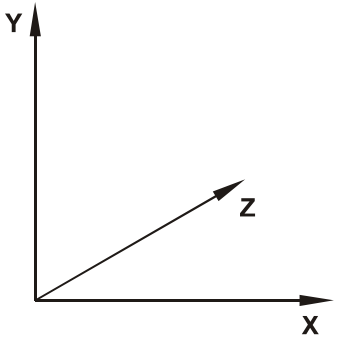

Рис. 3.3. Трехмерная система координат

Подобрать аналогию для трехмерной системы координат из окружающего мира несколько сложнее, чем для тех, которые мы уже рассмотрели, тем не менее, представить ее несложно (рис. 3.3). К имеющимся двум осям добавляется третья координатная ось — ось Z. В зависимости от ее положительного направления различают *левостороннюю* и *правостороннюю* систему трехмерных координат. В левосторонней системе координат при направлении оси X слева направо, а оси Y сверху вниз (как на рис. 3.2), положительное на-

правление оси Z будет направлено "от нас", то есть за плоскость рисунка. У правосторонней системы координат при тех же направлениях осей X и Y направление оси Z будет противоположным. Мы будем использовать левостороннюю систему координат, так как именно она применяется в Direct3D. Правосторонняя система координат применяется в других библиотеках трехмерной графики, например в OpenGL.

Совет

Для хранения координат точек и векторов в Direct3D есть удобные структуры: `D3DXVECTOR2` для двухмерных и `D3DXVECTOR3` для трехмерных координат. Использовать их можно, например, так:

```
// Запишем в переменную Vertex координаты точки (3, 4, 5)
D3DXVECTOR3 Vertex;
Vertex.x = 3.0f;
Vertex.y = 4.0f;
Vertex.z = 5.0f;
```

Как уже отмечалось, при программировании графики мы будем использовать двухмерные и трехмерные системы координат. Зачем нам еще и двухмерные координаты? Мы же программируем трехмерную графику?

Трехмерная система координат используется для хранения данных о виртуальном мире в памяти компьютера, двухмерная система — для непосредственного вывода на экран. Двухмерная система координат привязана ко всему экрану, если программа работает в полноэкранном режиме, либо к окну программы в оконном режиме. В любом случае начало координат находится в левом верхнем углу окна или экрана, ось X направлена вправо, а ось Y — вниз (рис. 3.4).

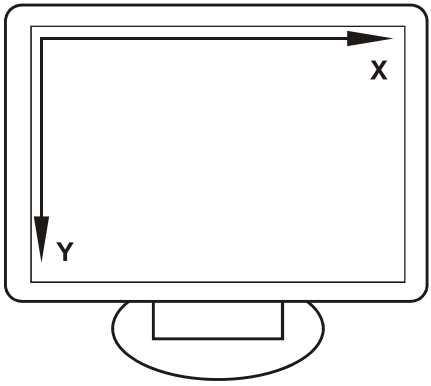

Рис. 3.4. Экранная система координат

Геометрические преобразования

С помощью систем координат можно описать положение точек в пространстве. С помощью нескольких точек можно получать отрезки и треугольники, называемые *примитивами*. С помощью примитивов уже можно строить неподвижные изображения. Для того чтобы получить анимированное изображение, необходимо каким-то образом от кадра к кадру изменять координаты примитивов. Все перемещения объектов в виртуальном пространстве выполняются с помощью трех основных геометрических преобразований: перемещения, вращения и масштабирования. Познакомимся с каждым из этих преобразований индивидуально.

Перемещение

Пожалуй, перемещение — это самое простое преобразование. С его помощью можно перенести вершины объекта с одного места на другое. Перемещая объект, мы как будто говорим ему, например: "Эй, сдвинься на три единицы вправо и на три единицы вверх". На рис. 3.5 показан треугольник до и после применения преобразования перемещения.

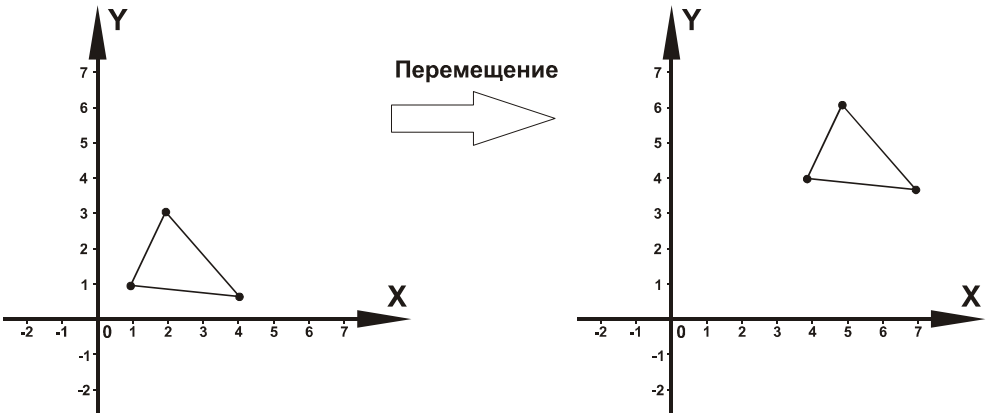

Рис. 3.5. Перемещение треугольника

Расчет координат при таком преобразовании совсем несложен: к каждой из координат нужно прибавить величину, на которую производится сдвиг. То есть просьба к объекту, приведенная выше, математически записывается так:

```
D3DXVECTOR2 Object(0.0f,0.0f);
Object.x +=3.0f;
Object.y +=3.0f;
```

Вращение

Преобразование вращения выполняет поворот объекта вокруг начала координат на указанное количество градусов. На рис. 3.6 в качестве примера изображен поворот треугольника.

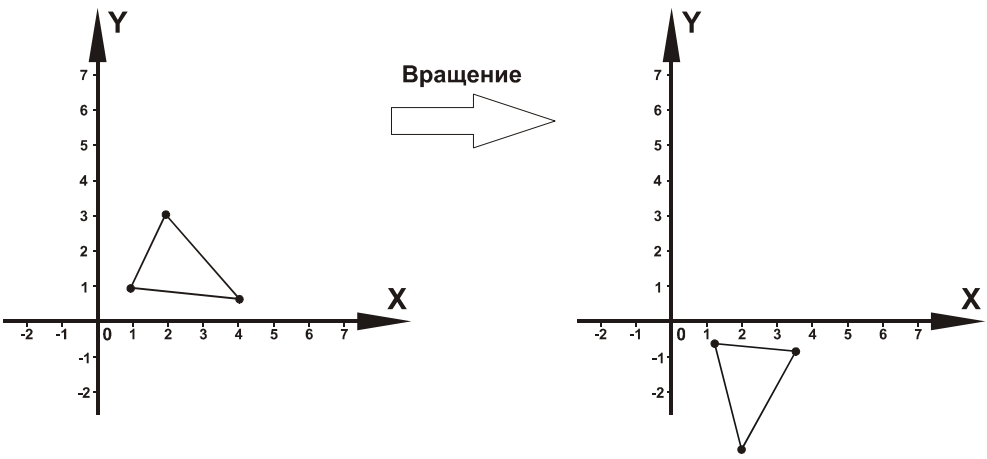

Рис. 3.6. Вращение треугольника

Важно помнить, что поворот всегда осуществляется вокруг начала координат против часовой стрелки (для поворота по часовой стрелке можно задать отрицательный угол поворота). Поворот объекта — это поворот всех его вершин. Координаты нового положения вершины при повороте на угол α рассчитываются следующим образом:

$$x = x \cdot \cos(\alpha) - y \cdot \sin(\alpha),$$
$$y = x \cdot \sin(\alpha) + y \cdot \cos(\alpha).$$

Приведем пример для вращения вершины с координатами (2,3) на угол 30°:

```
D3DXVECTOR2 Vertex(2.0f,3.0f);
Float Alfa = 30;
Vertex.x = Vertex.x*cos(Alfa)-Vertex.y*sin(Alfa);
Vertex.y = Vertex.x*sin(Alfa)+Vertex.y*cos(Alfa);
```

Замечание

Если быть до конца точным, то значение угла поворота следовало бы перевести в радианы, но ради простоты примера оставим все как есть. На всякий случай запомним, что для перевода улов из градусов в радианы имеется макрос `D3DXToRadian()` и макрос `D3DXToDegree()` для выполнения обратного перевода.

Масштабирование

Преобразование масштабирования позволяет уменьшать либо увеличивать размеры объекта в требуемое количество раз. В результате этого преобразования вершины, образующие объект, либо сдвигаются ближе друг к другу, если объект уменьшается, либо раздвигается дальше, если он увеличивается. На рис. 3.7 изображен треугольник до и после применения масштабирования.

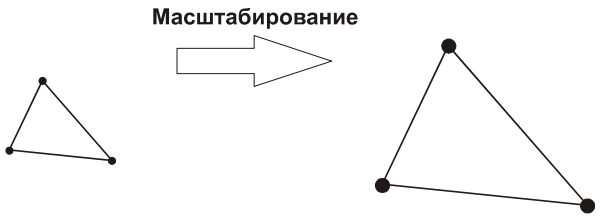

Рис. 3.7. Масштабирование треугольника

Чтобы получить координаты вершин отмасштабированного объекта, нужно умножить имеющиеся координаты на масштабный множитель, задающий новый размер объекта. Например, если имеется квадрат со стороной 2 и к нему применяется масштабирование с множителем 2, в результате получится квадрат со стороной 4. Если применить масштабирование с множителем 0,5, сторона получившегося квадрата будет равна 1. При использовании масштабного множителя, равного единице, размеры объекта не изменяются. Формула расчета координат при масштабировании с множителем M выглядит следующим образом:

$$x = x \cdot M,$$
$$y = y \cdot M.$$

В качестве примера рассмотрим, как программно реализуется вычисление координат вершины (2,3) при масштабировании с множителем 2:

```
D3DXVECTOR2 Vertex(2.0f,3.0f);
Float M = 2;
Vertex.x = Vertex.x*M;
Vertex.y = Vertex.y*M;
```

Векторы

Мы рассмотрели, как с помощью систем координат можно описывать положение точек в пространстве и как можно изменять положение объектов в пространстве с помощью геометрических преобразований. Все это позволяет нам

работать с положением тел в пространстве. Но кроме этого, при программировании трехмерной графики нам нужно будет иметь дело еще и с указанием направления. Например, как указать, в какую сторону направлена грань, в каком направлении идет луч света и как сориентирован наблюдатель в трехмерном пространстве? Именно для этого служат векторы.

Вектор представляет собой направленный отрезок. Координаты вектора записываются точно так же, как и координаты точки. Например, у вектора, изображенного на рис. 3.8, координаты такие: (4,6). Таким образом, можно отметить, что координаты вектора совпадают с координатами его конца (куда направлена стрелка) при условии, что начало вектора совпадает с точкой отсчета системы координат. Чтобы отличать обозначение вектора от обозначений обычных отрезков, над его именем добавляют стрелочку: $\vec{v}$.

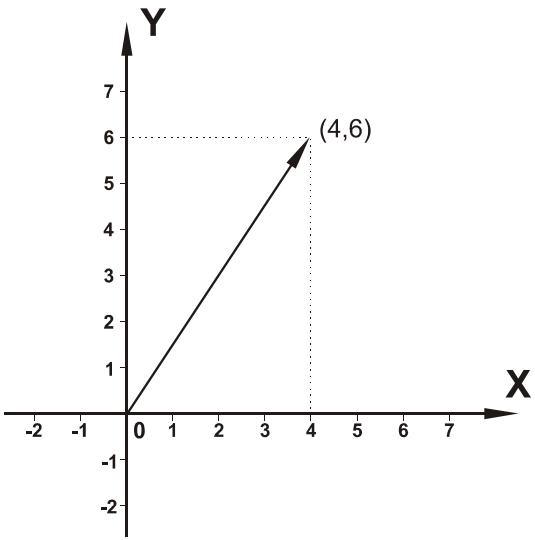

Рис. 3.8. Вектор и его координаты

Модуль вектора

Модуль вектора — это его длина. В математике модуль вектора обозначается следующим образом: $|\vec{v}|$. Как вычислить модуль вектора? Взглянем на рис. 3.9, из него видно, что вектор представляет собой гипотенузу в прямоугольном треугольнике, катеты в котором равны его соответствующим координатам.

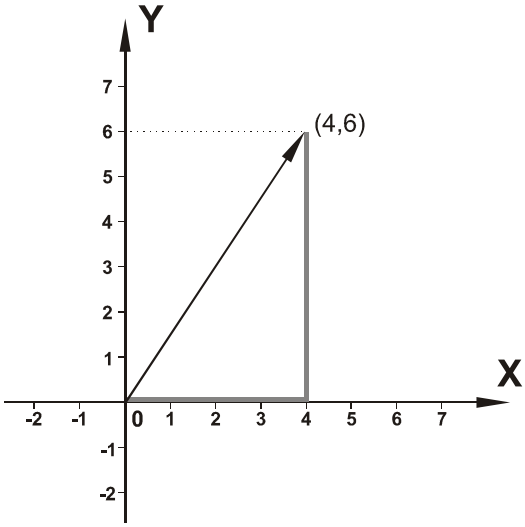

Рис. 3.9. Модуль вектора

Из этого легко заключить, что длину вектора можно вычислить по теореме Пифагора. Запишем выражение для модуля вектора на плоскости:

$$|\vec{v}| = \sqrt{x^2 + y^2} .$$

Для вектора в трехмерном пространстве модуль вычисляется аналогично, нужно лишь добавить в формулу координату z: $|\vec{v}| = \sqrt{x^2 + y^2 + z^2}$.

Программно вычисление модуля можно реализовать так:

```
D3DXVECTOR2 Vector(4.0f,6.0f);
float VectorMod = sqrt( Vector.x*Vector.x+Vector.y*Vector.y );
```

В Direct3D 10 нет специализированных функций для вычисления модуля вектора, так как такого рода вычисления выполняются шейдерами, их мы рассмотрим в *гл. 5*.

Сложение векторов

Как и над обычными числами, над векторами можно выполнять математические действия. Рассмотрим сложение векторов $\vec{v_1}$ и $\vec{v_2}$ (рис. 3.10).

Если складываемые векторы расположить так, что конец вектора $\vec{v_1}$ совпадает с началом вектора $\vec{v_2}$, то их суммарным вектором будет вектор $\vec{v_3}$, выходящий из начала вектора $\vec{v_1}$ и приходящий в конец вектора $\vec{v_2}$. Так осущест-

вляется сложение векторов графически, применительно же к координатам сложение векторов осуществляется путем суммирования их соответствующих координат:

$$x_3 = x_1 + x_2,$$
$$y_3 = y_1 + y_2.$$

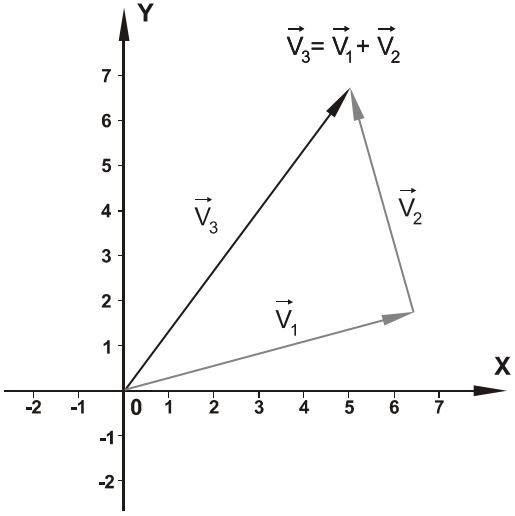

Рис. 3.10. Сложение векторов

Пример программного сложения векторов:

```
D3DXVECTOR2 Vector1(1.0f,2.0f);
D3DXVECTOR2 Vector2(3.0f,4.0f);
D3DXVECTOR2 Vector3(0.0f,0.0f);

Vector3.x = Vector1.x + Vector2.x;
Vector3.y = Vector1.y + Vector2.y;
```

или, что то же самое:

```
Vector3 = Vector1 + Vector2;
```

Вычитание векторов

Вычитание векторов $\vec{v_3} = \vec{v_1} - \vec{v_2}$ можно рассматривать как сложение, только у второго вектора ($\vec{v_2}$) нужно изменить направление на обратное. Рассмотрим разность векторов $\vec{v_1}$ и $\vec{v_2}$ (рис. 3.11).

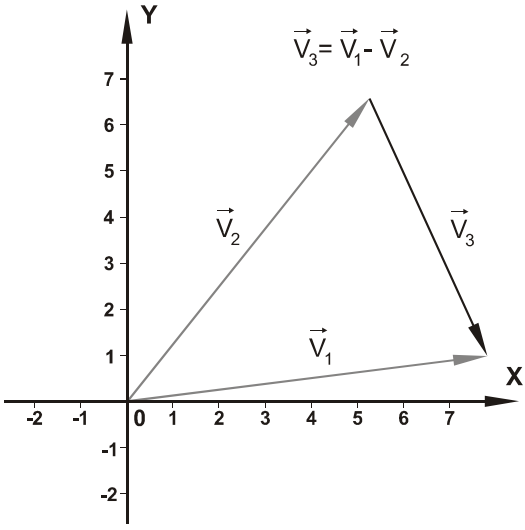

Рис. 3.11. Вычитание векторов

При вычитании из вектора $\vec{v_1}$ вектора $\vec{v_2}$ результирующий вектор $\vec{v_3}$ получается как вектор, выходящий из конца второго вектора в конец первого, при условии, что вектора $\vec{v_1}$ и $\vec{v_2}$ выходят из одной точки. Координаты результирующего вектора получаются вычитанием из соответствующих координат вектора $\vec{v_1}$ координат вектора $\vec{v_2}$. Пример программного вычисления разности векторов:

```
D3DXVECTOR2 Vector1(1.0f,2.0f);
D3DXVECTOR2 Vector2(3.0f,4.0f);
D3DXVECTOR2 Vector3(0.0f,0.0f);

Vector3.x = Vector1.x - Vector2.x;
Vector3.y = Vector1.y - Vector2.y;
```

или, что то же самое,

```
Vector3 = Vector1 - Vector2;
```

Обратите внимание, что здесь важна последовательность, из какого вектора какой вычитается: $\vec{v_1} - \vec{v_2} \neq \vec{v_2} - \vec{v_1}$.

Умножение вектора на число

Умножить вектор на число A — значит изменить его длину в A раз. Например, умножение вектора на 2 увеличивает его длину вдвое (рис. 3.12).

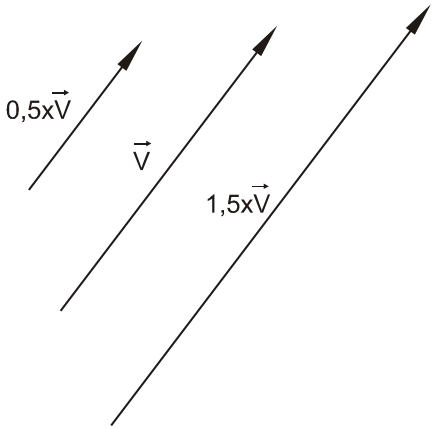

Рис. 3.12. Умножение вектора на число

Координаты вектора при умножении на число A вычисляются умножением всех координат на это число:

$$x = x \cdot A,$$
$$y = y \cdot A.$$

Пример программного умножения:

```
D3DXVECTOR2 Vector(1.0f,2.0f);
float Num = 1.5f;
Vector = Vector * Num;
```

Нормализация вектора

Нормализацией вектора называется изменение его длины на единичную при сохранении направления вектора. Нормализацию можно считать частным случаем умножения вектора на число. В этом случае в качестве числа A выступает значение, обратное модулю вектора: $A_{норм} = \frac{1}{|\vec{v}|}$.

Скалярное произведение векторов

В результате скалярного произведения двух векторов получается обычное число (скаляр). Численно результат скалярного произведения равен произведению модулей векторов на косинус угла между ними $\vec{v_1} \cdot \vec{v_2} = |\vec{v_1}| \cdot |\vec{v_2}| \cdot \cos(\alpha)$ (рис. 3.13).

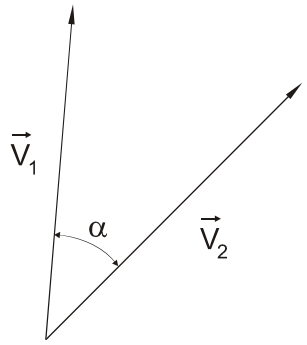

Рис. 3.13. Скалярное произведение векторов

Для нас скалярное произведение важно тем, что оно дает информацию об угле между векторами. Если модули векторов равны единице, результат скалярного произведения равен косинусу угла α. Это свойство используется, например, при расчете освещенности. Позднее, в *гл. 7*, мы к этому еще вернемся. С помощью координат векторов скалярное произведение рассчитывается так: $\vec{v_1} \cdot \vec{v_2} = (x_1 \cdot x_2) + (y_1 \cdot y_2) + (z_1 \cdot z_2)$. Нагляднее эту формулу демонстрирует программный пример:

```
D3DXVECTOR2 Vector1(1.0f,2.0f);
D3DXVECTOR2 Vector2(3.0f,4.0f);
float Scalar1_2 = (Vector1.x*Vector2.x)+(Vector1.y*Vector2.y)+
            (Vector1.z*Vector2.z);
```

Векторное произведение векторов

Как следует из названия действия, результатом векторного произведения вектора $\vec{v_1}$ на вектор $\vec{v_2}$ также является вектор $\vec{v_3}$. Этот вектор должен удовлетворять следующим условиям (рис. 3.14):

- Вектор $\vec{v_3}$ должен быть перпендикулярен к векторам $\vec{v_1}$ и $\vec{v_2}$.
- Вектора $\vec{v_1}$, $\vec{v_2}$ и $\vec{v_3}$ должны образовывать "правую тройку" векторов (если смотреть с конца вектора $\vec{v_3}$, кратчайший поворот от $\vec{v_1}$ к $\vec{v_2}$ должен происходить по часовой стрелке).
- Длина вектора $\vec{v_3}$ должна численно равняться площади параллелограмма, построенного на векторах $\vec{v_1}$ и $\vec{v_2}$ как на сторонах ($\left|\vec{v_3}\right| = \left|\vec{v_1}\right| \cdot \left|\vec{v_2}\right| \cdot \sin(\alpha)$).

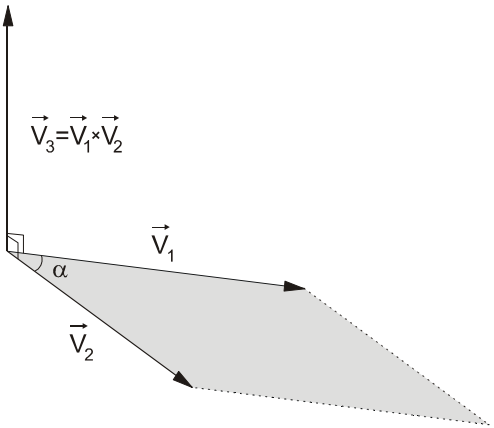

Рис. 3.14. Векторное произведение векторов

Для нас из всех этих условий интересно условие о перпендикулярности вектора $\vec{v_3}$ к двум другим. Это свойство можно использовать для расчета нормали к поверхности грани при известных координатах ее вершин. Приведем пример вычисления векторного произведения через координаты векторов:

```
D3DXVECTOR2 v1(1.0f,2.0f);
D3DXVECTOR2 v2(3.0f,4.0f);
D3DXVECTOR3 v3(0.0f,0.0f);

v3.x = v1.y*v2.z - v1.z*v2.y;
v3.y = v1.z*v2.x - v1.x*v2.z;
v3.z = v1.x*v2.y - v1.y*v2.x;
```

Как и в случае вычитания векторов, нужно иметь в виду, что векторное произведение некоммутативно, то есть результат зависит от перестановки умножаемых векторов: $\vec{v_1}\times\vec{v_2}\neq\vec{v_2}\times\vec{v_1}$.

Матрицы

Как мы скоро убедимся, матрицы находят широкое применение в программировании трехмерной графики. Все дело в том, что применение рассмотренных геометрических преобразований для решения насущных задач может быть затруднительно. Представим, что мы пишем компьютерную игру: гонки на автомобилях. Мы должны отслеживать положение каждой машины и перед выводом каждого кадра *по очереди* выполнять необходимые преобразования. То есть, когда машина меняет свое положение, мы должны для каждой

вершины выполнить преобразование перемещения, затем преобразование вращения и, наконец, масштабирования, если оно необходимо. При использовании матриц мы для всех этих преобразований используем одну матрицу, которой все они описываются. То есть новое положение машины получается простым умножением координат каждой вершины на итоговую матрицу преобразований. Как это делается на практике, мы рассмотрим ниже, а сейчас пока обсудим другой, не менее важный вопрос…

Что такое матрица?

Матрица представляет собой двухмерный массив чисел определенной размерности, ее также можно представить как набор векторов, которые "склеены" друг с другом. Вот так, например, выглядит матрица A размерностью 4×4:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}.$$

Обращение к элементу матрицы осуществляется путем указания номера строки и столбца нужного элемента. Например, обозначение a_{32} указывает на элемент, находящийся на пересечении третьей строки и второго столбца матрицы. Обратите внимание, что сначала указывается именно номер строки, а потом номер столбца.

В Direct3D 10 для работы с матрицами имеется специальная структура — `D3DXMATRIX`. Элементы матрицы в этом случае обозначаются так:

```
_11, _12, _13, _14
_21, _22, _23, _24
_31, _32, _33, _34
_41, _42, _43, _44
```

В качестве примера приведем фрагмент программы, в котором мы получаем доступ к элементам матрицы:

```
D3DXMATRIX Matrix;
Matrix._32 = 2.0f;
Matrix._44 = 3.0f;
```

Что представляет собой матрица, думаю, теперь ясно. Двигаемся дальше, нас ожидает знакомство с математическими действиями, которые можно выполнять над матрицами.

Сложение и вычитание матриц

Сложение матриц во многом аналогично сложению векторов. Мы просто складываем между собой соответствующие элементы. Единственным условием для выполнения матричного сложения является одинаковая размерность матриц. Вот пример сложения двух матриц размерности 2×2:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 6 & 8 \\ 10 & 12 \end{pmatrix}.$$

Вычитание матриц осуществляется аналогичным образом, только вместо сложения производится вычитание соответствующих элементов. Например,

$$\begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} - \begin{pmatrix} 1 & 10 \\ 2 & 11 \end{pmatrix} = \begin{pmatrix} 4 & -4 \\ 5 & -3 \end{pmatrix}.$$

В программе сложение и вычитание матриц осуществляется таким же образом, как сложение и вычитание переменных любого другого типа:

```
D3DXMATRIX ResMatrix;
D3DXMATRIX Matrix1;
D3DXMATRIX Matrix2;

ResMatrix = Matrix1 + Matrix2;
ResMatrix = Matrix2 - Matrix1;
```

Умножение матрицы на число

Как и при умножении вектора на число, для умножения на число матрицы нужно выполнить умножение для каждого элемента матрицы в отдельности. В качестве примера умножим матрицу 2×2 на число 5:

$$5 \cdot \begin{pmatrix} 1 & -2 \\ -3 & 4 \end{pmatrix} = \begin{pmatrix} 5 & -10 \\ -15 & 20 \end{pmatrix}.$$

При программировании умножение матрицы на число ничем не отличается от обычного умножения:

```
D3DXMATRIX ResMatrix;
D3DXMATRIX Matrix1;
float Num = 3.0f;

ResMatrix = Matrix1*Num;
```

Умножение матрицы на матрицу

При умножении двух матриц (например, A и B) должно выполняться обязательное условие: число столбцов первой матрицы (A) должно совпадать с числом строк второй матрицы (B). Само произведение выполняется по такому правилу: элемент c_{ij} результирующей матрицы C равен скалярному произведению i-го вектора-стоки матрицы A и j-го вектора-столбца матрицы B. Непонятно? Давайте рассмотрим умножение матрицы A на матрицу B в общем виде:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}, \qquad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{pmatrix},$$

$$A \cdot B = C = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} \end{pmatrix}.$$

В результирующей матрице будет столько же строк, сколько в первом сомножителе, и столько же столбцов, сколько их во втором сомножителе. По правилу умножения элемент c_{11} матрицы C равен скалярному произведению первого вектора-строки (то есть первой строки) матрицы A и первого вектора-столбца (первого столбца) матрицы B. Вспоминаем, как вычисляется скалярное произведение, и записываем его применительно к нашему случаю. Остальные элементы результирующей матрицы получаются аналогичным образом: как значение скалярного произведения соответствующей строки первой матрицы и столбца второй.

Остается добавить, что матричное умножение также некоммутативно, результат меняется от перемены перемножаемых матриц местами. А в некоторых случаях результат матричного умножения при перемене мест сомножителей не определен вообще. В качестве примера можно привести рассмотренные матрицы A и B: результат умножения $A \cdot B$ определен (и мы его получили), а вот перемножить матрицы как $B \cdot A$ не получится, так как число столбцов матрицы B составляет 2 и не равно числу строк матрицы A, она имеет три строки.

В программе умножение матрицы на матрицу можно осуществлять привычным способом, как умножение других переменных, или же с помощью специальной функции. Вот так осуществляется обычное умножение:

```
D3DXMATRIX ResMatrix;
```

```
D3DXMATRIX Matrix1;
D3DXMATRIX Matrix1;

ResMatrix = Matrix1 * Matrix2;
```

Для перемножения двух матриц используется функция `D3DXMatrixMultiply()`, ей нужно передать три параметра: указатель на результирующую матрицу, указатель на первую матрицу и указатель на вторую матрицу для умножения. То же самое умножение с помощью функции записывается так:

```
D3DXMATRIX ResMatrix;
D3DXMATRIX Matrix1;
D3DXMATRIX Matrix1;

D3DXMatrixMultiply(&ResMatrix, &Matrix1, &Matrix2);
```

Единичная матрица

Единичной матрицей называется матрица, у которой все элементы, лежащие на главной диагонали, равны единице, а остальные элементы равны нулю (главная диагональ матрицы — это диагональ элементов с одинаковыми индексами: a_{11}, a_{22}, a_{33}, …, a_{nn}). Вот как выглядит единичная матрица размерностью 4×4:

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

Единичная матрица обладает важным свойством: при матричном умножении на единичную матрицу в результате получается исходная матрица, то есть $M \cdot E = E \cdot M = M$. Нам это свойство пригодится для инициализации переменных типа `D3DXMATRIX`, после чего их можно будет использовать для умножения с другими матрицами. В Direct3D 10 имеется специальная функция для "превращения" указанной матрицы в единичную. Эту работу выполняет функция `D3DXMatrixIdentity()`, для этого ей нужно передать только один параметр: указатель на структуру `D3DXMATRIX`, в которую требуется занести значения единичной матрицы. Вот пример, иллюстрирующий ее применение:

```
D3DXMATRIX EMatrix;
D3DXMatrixIdentity(&EMatrix);
```

Матрицы геометрических преобразований

Что такое матрицы и как выполняются действия над ними, мы рассмотрели. Теперь мы можем познакомиться с тем, как с их помощью реализуются геометрические преобразования, и в чем именно состоит преимущество такого подхода.

Матрица перемещения

Как осуществляется преобразование перемещения, мы уже знаем. С помощью матрицы можно достичь такого же результата. Матрица перемещения выглядит следующим образом:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ D_x & D_y & D_z & 1 \end{pmatrix},$$

где D_x, D_y и D_z представляют собой смещения нового положения вершины относительно исходного вдоль координатных осей X, Y и Z соответственно. Конечно же, нет никакой необходимости создавать в программе эту матрицу вручную, всю работу сделает функция `D3DXMatrixTranslation()`. Ей требуется передать параметрами указатель на результирующую матрицу и смещения вдоль координатных осей. Если нам, например, необходима матрица для перемещения вершины на 2 единицы вдоль оси X, на 3 единицы вдоль оси Y и на 4 единицы вдоль оси Z, то ее можно получить с помощью следующего кода:

```
D3DXMATRIX ResMatrix;
D3DXVECTOR3 Offset (2.0f, 3.0f, 4.0f);


D3DXMatrixTranslation(&ResMatrix, Offset.x, Offset.y, Offset.z);
```

Матрица вращения

С помощью матриц можно осуществлять и преобразование вращения. В трехмерном пространстве вращение объекта в самом общем виде представляют как комбинацию вращений вокруг координатных осей: вокруг оси X, Y и Z.

Для вращения вокруг каждой из осей существует своя отдельная матрица. Для вращения вокруг оси X матрица имеет следующий вид:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

где θ — это угол вращения, он задается в радианах. Отсчет угла производится по часовой стрелке, если смотреть вдоль выбранной оси, со стороны положительного направления, на начало координат.

Матрица для вращения вокруг оси Y выглядит немного по-другому:

$$\begin{pmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

И, наконец, вот как выглядит матрица для вращения вокруг оси Z:

$$\begin{pmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

В Direct3D 10 имеются специальные функции для создания матриц вращения вокруг осей X, Y, и Z: это `D3DXMatrixRotationX()`, `D3DXMatrixRotationY()`, и `D3DXMatrixRotationZ()` соответственно. Все эти функции требуют указания двух параметров: указателя на результирующую матрицу и значения угла вращения. Вот как выглядит отрывок программы, в котором создается матрица вращения вокруг оси X на угол в 30° (обратите внимание, что значение угла переводится из градусов в радианы с помощью макроса `D3DXToRadian()`):

```
D3DXMATRIX ResMatrix;
float Angle = D3DXToRadian(30);

D3DXMatrixRotationX(&ResMatrix, Angle);
```

Матрица масштабирования

Это последняя нерассмотренная матрица из матриц преобразований. По воздействию на координаты она идентична рассмотренным выше уравнениям

для масштабирования координат точки. Сама она выглядит следующим образом:

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

S_x, S_y и S_z представляют собой коэффициенты масштабирования вдоль соответствующих координатных осей. Для создания такой матрицы в программе существует функция `D3DXMatrixScaling()`, в качестве параметров которой требуется задать указатель на результирующую матрицу и коэффициенты масштабирования по осям X, Y, и Z. В качестве примера рассмотрим создание матрицы, когда необходимо объект "сплющить" вдоль оси X с коэффициентом 0,8, растянуть вдоль оси Y с коэффициентом 1,2, а масштаб по оси Z оставить без изменений:

```
D3DXMATRIX ResMatrix;


D3DXMatrixScaling(&ResMatrix, 0.8f, 1.2f, 1.0f);
```

Комбинирование нескольких преобразований

Согласитесь, действия с матрицами — штука не очень простая. Во всяком случае, выглядит сложнее, чем обычные уравнения для геометрических преобразований. Так, все-таки, дает ли использование матриц какие-нибудь преимущества? Безусловно, да. И мы уже коротко об этом упоминали. Речь идет о возможности "накопить" в себе несколько геометрических преобразований и затем применить их все разом, выполнив лишь умножение координат точки на одну эту матрицу. Достигается все это обычным перемножением матриц, преобразования которых нужно совместить. Здесь нужно напомнить, что важен порядок, в котором матрицы перемножаются. Умножать матрицы следует в том порядке, в котором преобразования применяются к координатам.

Матрицы в Direct3D 10

Мы рассмотрели матрицы для выполнения геометрических преобразований. Для того чтобы получить изображение трехмерной сцены на экране, нам еще нужно рассмотреть три матрицы, связанные непосредственно с Direct3D. Чтобы сразу понять, о чем пойдет речь, проведем аналогию между получением

изображения трехмерной виртуальной сцены и получением изображения с помощью фотоаппарата. Итак, какую последовательность действий нужно выполнить, чтобы создать условия для съемки и получения фотографии? Сначала необходимо нужным образом расположить снимаемые объекты в пространстве, после этого выбрать точку съемки, в которой размещаем фотоаппарат, и, наконец, подобрать объектив для фотоаппарата, чтобы получить желаемый вид объектов. Получение изображения трехмерной сцены с помощью Direct3D во многом похоже на описанный процесс, и все эти действия выполняются с помощью трех матриц. Вот они:

- мировая матрица указывает, как располагать объекты в пространстве трехмерной сцены ("мировом" пространстве), для этого используются матрицы геометрических преобразований;
- матрица вида задает координаты точки, в которой находится виртуальная камера и в какую точку она направлена, полученное изображение и будет являться "снимком" сцены с помощью этой камеры;
- матрица проекции определяет, как будет происходить преобразование (проецирование) трехмерной сцены в двухмерное изображение на экране.

Теперь поговорим обо всех этих матрицах подробнее.

Мировая матрица

Как мы выяснили, с помощью этой матрицы мы располагаем трехмерные объекты в пространстве. Еще это можно представить как расстановку виртуальной "мебели" и декораций (рис. 3.15).

То есть с помощью мировой матрицы мы получаем координаты исходных вершин объекта в "мировой" системе координат. Каким образом это делается в программе? Предположим, что у нас для объекта созданы следующие матрицы преобразований: перемещения, вращения и масштабирования (имена переменных `mTrans`, `mRot` и `mScale` соответственно). Наша задача — получить для этого объекта мировую матрицу. Выполним это следующим образом. Сначала объявим и инициализируем переменную `mWorldMatrix`, а затем присвоим ей результат умножения матриц преобразования:

```
D3DXMATRIX mWorldMatrix;
D3DXMatrixIdentity(&mWorldMatrix);
mWorldMatrix = mRot*mTrans*mScale;
```

Эта матрица передается для дальнейших вычислений в вершинный шейдер. Мы еще будем говорить об этом в *гл. 5*.

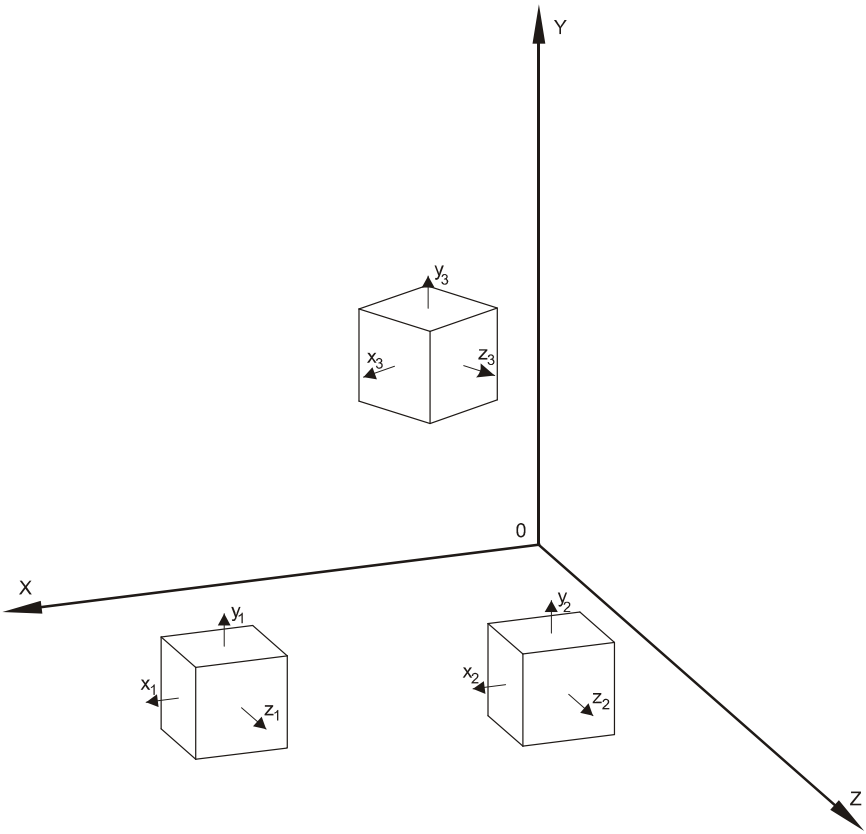

Рис. 3.15. Размещение объектов в виртуальном пространстве с помощью мировой матрицы

Может возникнуть вопрос: почему порядок перемножения матриц преобразований выбран именно таким? Если в двух словах — для предсказуемости результата. Разберем конкретный пример. Представим, что в нашей сцене есть столб, возвышающийся из точки начала мировых координат (вдоль оси Y) и на некотором удалении от него двигается танк (на плоскости XZ). Танк остановился и разворачивается на месте. Какие матрицы преобразований и в какой последовательности использовать для моделирования этого процесса? Очевидно, что нужна матрица перемещения танка на "некоторое удаление" от столба (начала координат), нужна матрица вращения (танк разворачивается на месте — вращается вокруг оси Y); применяется ли к танкам масштабирование, мне неизвестно, в нашем случае оно не нужно, и это лишь упростит объяснение. В каком же порядке перемножать? Если мы остановимся на предложенном варианте и первой поставим матрицу вращения,

то потом, с каждым кадром увеличивая угол вращения, будем наблюдать разворачивающийся на месте танк. Как, в общем-то, и было задумано. Но попробуйте поменять матрицы преобразований местами. Танк будет вращаться вокруг столба, словно бы он вдруг решил покататься на карусели. И этого вполне следовало ожидать. В первом случае танк сначала поворачивался вокруг оси Y, находясь при этом в начале координат, а потом переносился на нужное место. Во втором случае мы сначала ставили его в нужную позицию, а затем поворачивали вокруг оси Y. Вот такая карусель… Из этого можно сделать вывод: будьте внимательны при составлении мировой матрицы.

Матрица вида

Матрица вида контролирует положение камеры в трехмерном пространстве, это своего рода мировая матрица для камеры (рис. 3.16).

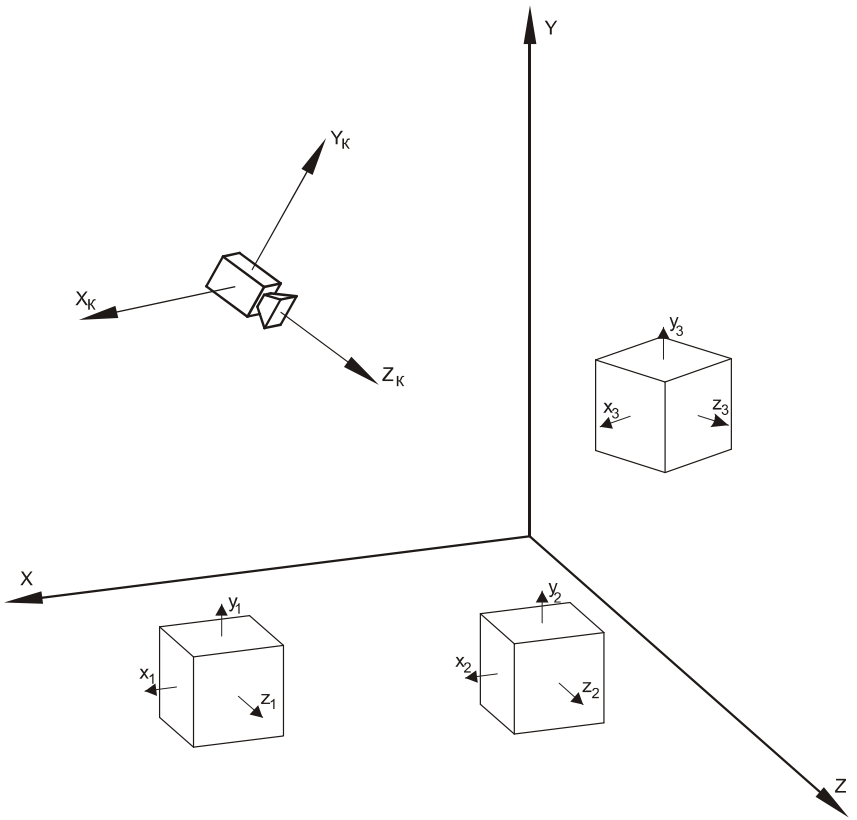

Рис. 3.16. Расположение камеры в "мировом" пространстве с помощью матрицы вида

Для создания матрицы вида существует специальная функция `D3DXMatrixLookAtLH()`, имеющая следующий прототип:

```
D3DXMatrixLookAtLHD3DXMATRIX * D3DXMatrixLookAtLH(
  D3DXMATRIX *pOut,
  CONST D3DXVECTOR3 *pEye,
  CONST D3DXVECTOR3 *pAt,
  CONST D3DXVECTOR3 *pUp
);
```

Для ее вызова функции требуется передать следующие параметры:

- `pOut` — указатель на результирующую матрицу;
- `pEye` — указатель на структуру, содержащую координаты положения камеры в "мировом" пространстве;
- `pAt` — указатель на структуру, содержащую координаты точки, на которую направлена камера;
- `pUp` — указатель на структуру, содержащую координаты вектора, задающего направление вверх от камеры (он нужен для проведения расчетов и для гарантии правильной ориентации полученного изображения).

Буквы "LH" в названии функции означают "left-handed", левосторонняя система координат. Мы уже говорили, что в Direct3D используется именно левосторонняя система координат, если по каким-то причинам нужно использовать правостороннюю систему координат, существует аналогичная функция `D3DXMatrixLookAtRH()`. Приведем пример использования функции для получения матрицы вида:

```
D3DXMATRIX mViewMatrix;
D3DXVECTOR3 Eye(0.0f, -3.0f, 0.0f);
D3DXVECTOR3 LookAt(0.0f, 0.0f, 0.0f);
D3DXVECTOR3 Up(0.0f, 1.0f, 0.0f);


D3DXMatrixLookAtLH(&mViewMatrix, &Eye, &LookAt, &Up);
```

Замечание

Хоть мы и говорили о положении камеры в пространстве, в действительности все немного сложнее. На самом деле, создается особая система координат, связанная с камерой. В этой системе координат камера всегда находится в начале координат (рис. 3.17) и направлена в положительном направлении оси Z. Когда мы изменяем положение камеры, фактически меняется положение объектов сцены в системе координат, связанной с камерой. Для нас, быть может, эта особенность не будет иметь принципиального значения, но знать, "какое оно там внутри", всегда полезно.

Матрица проекции

Матрица проекции, как мы выяснили, отвечает за правильное преобразование трехмерной сцены в итоговое изображение. Способы проецирования могут быть как с коррекцией перспективы (объекты уменьшаются по мере удаления от камеры), так и без нее. Проекции без коррекции перспективы могут использоваться в системах автоматизированного проектирования, но для нас они пока интереса не представляют. Мы ведь хотим повторить окружающий нас мир, значит, придется иметь дело с перспективой. В этом случае мы всего лишь должны выбрать для составления матрицы проекции соответствующую функцию. Рассмотрим прототип функции `D3DXMatrixPerspectiveFovLH()`:

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

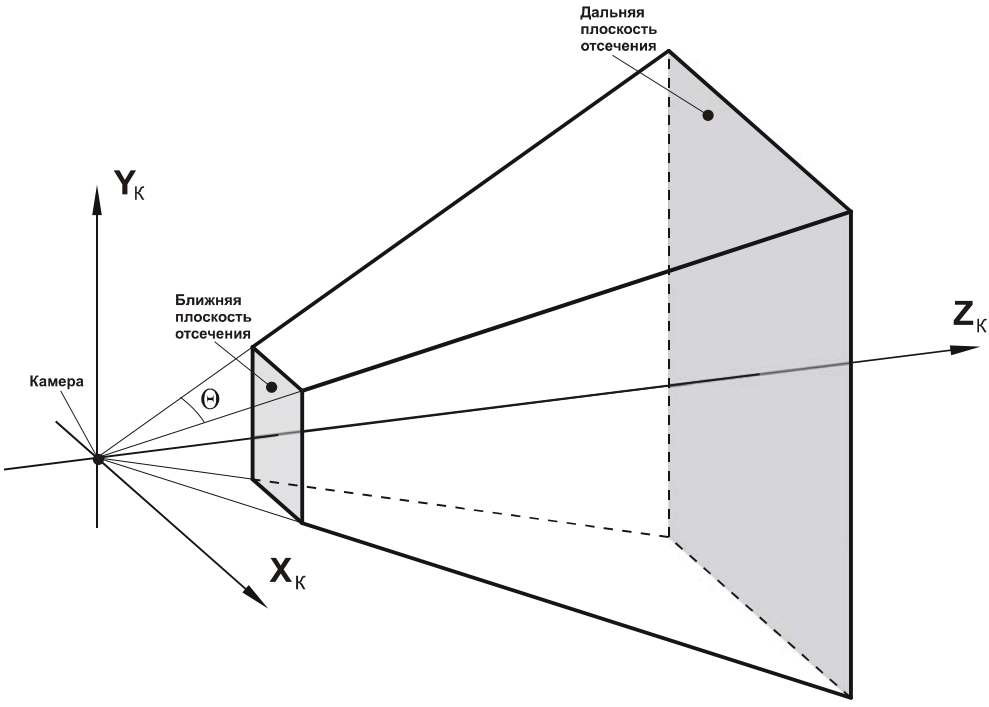

Рис. 3.17. Параметры для составления матрицы проекции

Параметры у этой функции следующие (рис. 3.17):

- `pOut` — указатель на результирующую матрицу;
- `fovy` — угол, ограничивающий поле зрения в направлении оси Y, указывается в радианах. Мы всегда будем использовать значение `D3DX_PI/4` (45º);
- `Aspect` — соотношение сторон окна либо экрана, если изображение выводится в полноэкранном режиме;
- `zn` — координата Z ближней плоскости отсечения;
- `zf` — координата Z дальней плоскости отсечения, объекты, находящиеся за ней, считаются расположенными за горизонтом и не отображаются.

Мы познакомились с основами математики для программирования трехмерной графики. Теперь мы можем шагнуть на следующую ступеньку, где мы, наконец-то, выведем на экран более или менее осмысленное изображение.

Глава 4

Двухмерная графика

В предыдущей главе мы ознакомились с элементами математики, применяемыми для трехмерной графики. Однако одна математика без программирования нам не поможет сколько-нибудь продвинуться по пути изучения Direct3D 10. Нам необходима практика. Перед тем как мы погрузимся в мир трехмерной графики, попробуем свои силы с графикой двухмерной. Программа, выполняющая создание устройства Direct3D, у нас уже написана, но пока она показывает лишь пустой экран, что не слишком интересно. Пусть первое, что мы научимся выводить на экран, будет текст. Выводить надписи на экран для любой программы очень полезно, независимо от того, игра это, расчетная программа или система автоматизированного проектирования. Кроме того, выводить на экран текст в Direct3D 10, наверное, проще всего. Приступаем…

Вывод текста

Организация вывода текста в Direct3D 10 во многом похожа на организацию вывода текста стандартными средствами Windows. Сначала требуется создать шрифтовой объект, который содержит информацию о том, как будут выглядеть символы, затем в функции визуализации вызвать метод `DrawText` шрифтового объекта для рисования текста во вторичный буфер. Итак, для того чтобы иметь возможность выводить текст, нам придется внести в нашу первую программу (см. листинг 2.2) следующие изменения:

- ввести новую глобальную переменную `g_pFont` — указатель на шрифтовой объект;
- добавить в функцию инициализации устройства Direct3D 10 строку с вызовом функции `D3DX10CreateFont()` для создания шрифтового объекта;
- в функцию прорисовки трехмерной сцены внести вызов метода `DrawText` объекта `g_pFont`;

- внести изменения в функцию освобождения памяти `Cleanup()` для корректного освобождения указателя `g_pFont`.

Как видно, для появления в нашей программе возможности вывода текста на экран нужно добавить в нее всего четыре строчки.

Про шрифтовой объект, наверное, необходимо сказать несколько слов. Объект представляет собой указатель на интерфейс `ID3DX10FONT`. Этот объект не просто определяет вид и начертание символов, у него имеются ресурсы и методы, необходимые для вывода текста указанным шрифтом через определенное нами устройство Direct3D.

Перейдем к составлению программы. Создадим новый проект и скопируем в него исходный текст из минимального приложения Direct3D 10 (см. листинг 2.2). В первую очередь введем новую глобальную переменную. Список глобальных переменных должен выглядеть следующим образом:

```
HWND        g_hWnd = NULL;
D3D10_DRIVER_TYPE       g_driverType = D3D10_DRIVER_TYPE_NULL;
ID3D10Device*           g_pd3dDevice = NULL;
IDXGISwapChain*         g_pSwapChain = NULL;
ID3D10RenderTargetView* g_pRenderTargetView = NULL;


// Указатель на шрифтовой объект
ID3DX10FONT*            g_pFont = NULL;
```

> ***Замечание***
>
> Для объявления указателя на объект шрифта можно также пользоваться типом `LPD3DX10FONT`, он описан как указатель на интерфейс `ID3DX10FONT`.

Следующим шагом станет изменение функции инициализации устройства Direct3D. Откроем функцию `InitDirect3D10()` и в самый конец добавим строку:

```
D3DX10CreateFont( g_pd3dDevice, 14, 8, 1, 1, FALSE, 0, 0, 0,
                  DEFAULT_PITCH|FF_MODERN, L"Verdana", &g_pFont );
```

В этой строке вызывается функция `D3DX10CreateFont()`, имеющая следующий прототип:

```
HRESULT D3DX10CreateFont(
  ID3D10Device *pDevice,
  INT Height,
  UINT Width,
  UINT Weight,
  UINT MipLevels,
```

```
  BOOL Italic,
  UINT CharSet,
  UINT OutputPrecision,
  UINT Quality,
  UINT PitchAndFamily,
  LPCSTR pFaceName,
  LPD3DX10FONT *ppFont
);
```

Функции надо передать следующие параметры:

- `pDevice` — указатель на интерфейс устройства (типа `ID3D10Device`), с которым будет связан шрифтовой объект;
- `Height` и `Width` — высота и ширина символов шрифта в логических единицах (в нашем случае логическая единица соответствует пикселу экрана);
- `Weight` — толщина линий символа шрифта;
- `MipLevels` — количество уровней мипмаппинга (множественного отображения, MIP mapping, т. е. использования для объектов одной и той же текстуры с разным разрешением для его отображения на разных расстояниях от камеры);
- `Italic` — флаг, указывающий, содержит ли шрифт символы наклонного начертания (italic), если да — следует указать значение `TRUE`, если нет — значение `FALSE`;
- `CharSet` — код символьного набора шрифта;
- `OutputPrecision` — поле, определяющее, каким образом система Windows должна согласовывать размеры и характеристики существующих шрифтов с параметрами, указанным в функции;
- `Quality` — поле, указывающее, как система Windows должна согласовывать желаемый и реально используемый шрифт, которое применяется только к растровым шрифтам и не должно использоваться для шрифтов TrueType;
- `PitchAndFamily` — флаг, содержащий информацию о шаге символов в строке и о принадлежности шрифта к тому или иному семейству (то есть, например, будет ли шрифт иметь фиксированную или переменную ширину символов);
- `pFaceName` — строка, указывающая имя желаемого шрифта;
- `ppFont` — выходной параметр, в котором функция возвращает указатель на интерфейс `ID3D10Device`, представляющий собой созданный шрифтовой объект.

Замечание

Подробную информацию о параметрах этой и других функций можно найти в *Приложении 1*.

Полностью функция `InitDirect3D10()` должна выглядеть следующим образом:

```
HRESULT InitDirect3D10()
{
     HRESULT hr = S_OK;

     // Узнаем размеры клиентской области окна
  RECT rc;
  GetClientRect( g_hWnd, &rc );
  UINT width = rc.right - rc.left;
  UINT height = rc.bottom - rc.top;

     // Список возможных типов устройства
     D3D10_DRIVER_TYPE driverTypes[] =
  {
      D3D10_DRIVER_TYPE_HARDWARE,
      D3D10_DRIVER_TYPE_REFERENCE,
  };
  UINT numDriverTypes = sizeof(driverTypes) / sizeof(driverTypes[0]);

     // Заполняем структуру
  DXGI_SWAP_CHAIN_DESC sd;
  ZeroMemory( &sd, sizeof(sd) );
  sd.BufferCount = 1;
  sd.BufferDesc.Width = width;
  sd.BufferDesc.Height = height;
  sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
  sd.BufferDesc.RefreshRate.Numerator = 60;
  sd.BufferDesc.RefreshRate.Denominator = 1;
  sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
  sd.OutputWindow = g_hWnd;
  sd.SampleDesc.Count = 1;
  sd.SampleDesc.Quality = 0;
  sd.Windowed = TRUE;

  // Пытаемся создать устройство, проходя по списку
  // Как только получилось - выходим из цикла
```

```
for( UINT driverTypeIndex = 0; driverTypeIndex < numDriverTypes;
             driverTypeIndex++ )
{
   g_driverType = driverTypes[driverTypeIndex];
   hr = D3D10CreateDeviceAndSwapChain( NULL, g_driverType, NULL, 0,
              D3D10_SDK_VERSION, &sd, &g_pSwapChain, &g_pd3dDevice );
   if( SUCCEEDED( hr ) )
     break;
}
if( FAILED(hr) )
       return hr;

// Представление данных для
// буфера визуализации
ID3D10Texture2D *pBackBuffer;
// Получим доступ к вторичному буферу с индексом 0
hr = g_pSwapChain->GetBuffer( 0, __uuidof( ID3D10Texture2D ),
                              (LPVOID*)&pBackBuffer );
if( FAILED(hr) )
      return hr;
// Создадим представление данных
hr = g_pd3dDevice->CreateRenderTargetView( pBackBuffer, NULL,
                                           &g_pRenderTargetView );
pBackBuffer->Release();
if( FAILED(hr) )
      return hr;

g_pd3dDevice->OMSetRenderTargets( 1, &g_pRenderTargetView, NULL );

// Настроим область отображения
D3D10_VIEWPORT vp;
vp.Width = width;
vp.Height = height;
vp.MinDepth = 0.0f;
vp.MaxDepth = 1.0f;
vp.TopLeftX = 0;
vp.TopLeftY = 0;
g_pd3dDevice->RSSetViewports( 1, &vp );
```

```
  // Создаем шрифтовой объект
  D3DX10CreateFont( g_pd3dDevice, 14, 8, 1, 1, FALSE, 0, 0, 0,
                    DEFAULT_PITCH|FF_MODERN, L"Verdana", &g_pFont );

  return S_OK;
}
```

После выполнения этой функции у нас будет создан шрифтовой объект, и с его помощью мы сможем вывести текст на экран. Открываем функцию `RenderScene()`, добавляем после закрашивания вторичного буфера следующие строки:

```
//Размеры прямоугольника для форматирования текста
RECT Rect;
Rect.left=10;
Rect.top=10;
Rect.right=600;
Rect.bottom=380;

g_pFont->DrawText(NULL, L"Этот текст мы вывели на экран", -1, &Rect,
          DT_CENTER | DT_VCENTER , D3DXCOLOR(1.0,1.0,1.0,1.0));
```

Со структурой `RECT` мы уже сталкивались, здесь мы сохраняем в ней параметры прямоугольника, относительно которого будет происходить форматирование текста, после этого мы вызываем метод `DrawText` шрифтового объекта. Чтобы разобраться со всеми параметрами, рассмотрим прототип метода:

```
INT DrawText(
  ID3DX10Sprite *pSprite,
  LPCSTR pString,
  INT Count,
  LPRECT pRect,
  UINT Format,
  D3DXCOLOR Color
);
```

Для вызова метода необходимо передать следующие параметры:

- `pSprite` — указатель на спрайтовый объект `ID3DX10Sprite`, который будет отображать строку символов. Может иметь значение `NULL`, в этом случае Direct3D выведет строку символов, используя собственный спрайтовый объект;
- `pString` — указатель на строку символов, которую необходимо вывести;
- `Count` — количество символов в выводимой строке, можно поставить значение –1, в этом случае будет считаться, что `pString` указывает на строку

с завершающим нулем и количество символов будет подсчитано автоматически;

- `pRect` — указатель на прямоугольник, относительно которого осуществляется форматирование текста;
- `Format` — флаг, который определяет способ форматирования текста, может представлять собой любую комбинацию специальных флагов, которые приведены в табл. 4.1;
- `Color` — цвет выводимого текста.

В случае успешного выполнения метод возвращает высоту текста в пикселах. Если указан флаг `DT_VCENTER` либо `DT_BOTTOM`, возвращаемое значение представляет собой смещение строки текста относительно указанного прямоугольника для форматирования (от верхней границы прямоугольника до нижней границы текста). В случае неудачного завершения метод возвращает нулевое значение.

Таблица 4.1. *Флаги для указания способа форматирования текста*

Значение	Описание
`DT_BOTTOM`	Выравнивает текст по нижней границе прямоугольника, нужно использовать в сочетании с флагом `DT_SINGLELINE`
`DT_CALCRECT`	Автоматически вычисляет ширину и высоту прямоугольника, исходя из длины указанной строки для вывода. Если выводится многострочный текст, `ID3DX10Font::DrawText` сохранит ширину прямоугольника, заданного параметром `pRect`, и изменит его высоту так, чтобы он охватывал последнюю строку текста. В случае же, если текст представляет собой единственную строку, метод поменяет правую границу прямоугольника таким образом, чтобы последний символ строки находился внутри прямоугольника. В обоих случаях `ID3DX10Font::DrawText` возвращает высоту отформатированного текста, но вывод текста не осуществляет
`DT_CENTER`	Выравнивает текст в горизонтальном направлении по центру прямоугольника
`DT_EXPANDTABS`	Заменяет символы табуляции пробелами. По умолчанию один символ табуляции эквивалентен восьми пробелам
`DT_LEFT`	Выравнивает текст по левому краю прямоугольника
`DT_NOCLIP`	Рисует текст без отсечения, при использовании этого флага метод работает несколько быстрее
`DT_RIGHT`	Выравнивает текст по правому краю прямоугольника

Таблица 4.1 *(окончание)*

Значение	Описание
DT_RTLREADING	Выводит текст в режиме "справа налево", если выбран соответствующий шрифт (арабский или еврейский), используется для вывода двунаправленных текстов. По умолчанию направление вывода всего остального текста — "слева направо"
DT_SINGLELINE	Отображает весь текст в одну строку. Имеющиеся в тексте символы возврата каретки и перевода строки не разрывают строку
DT_TOP	Выравнивает текст по верхней границе прямоугольника
DT_VCENTER	Выравнивает текст по центру прямоугольника в вертикальном направлении (только для текста в одну строку)
DT_WORDBREAK	Выполняет перенос по словам. Очередное слово в тексте автоматически переносится на следующую строку, если в текущей строке оно может выйти за границу прямоугольника, заданного параметром pRect. Имеющиеся в тексте символы возврата каретки и перевода строки также разрывают строку

Обратите внимание, что для установки компонентов цвета используется структура D3DXCOLOR, которая описывается следующим образом:

```
typedef struct D3DXCOLOR {
    FLOAT r;
    FLOAT g;
    FLOAT b;
    FLOAT a;
} D3DXCOLOR, *LPD3DXCOLOR;
```

Структура позволяет задать компоненты цвета (красный, зеленый, синий, прозрачность) в виде четырех значений с плавающей точкой, лежащих в интервале от 0 до 1.

При вызове метода ID3DX10Font::DrawText мы используем параметры, смысл которых можно передать так: отобразить текст "Этот текст мы вывели на экран" в прямоугольнике с параметрами, содержащимися в переменной Rect, выполнить выравнивание текста по центру в вертикальном и горизонтальном направлениях, использовать белый цвет символов (все компоненты равны 1). Для того чтобы увидеть текст на экране, нам остается лишь вызвать метод Present цепочки переключения:

```
g_pSwapChain->Present( 0, 0 );
```

Полностью функция `RenderScene()` должна выглядеть следующим образом:

```
void RenderScene()
{
    // Очищаем вторичный буфер
    //(компоненты красного, зеленого, синего, прозрачность)
    float ClearColor[4] = { 0.1f, 0.5f, 0.1f, 1.0f };
    g_pd3dDevice->ClearRenderTargetView( g_pRenderTargetView,
                                         ClearColor );

    //Размеры прямоугольника для форматирования текста
    RECT Rect;
    Rect.left=10;
    Rect.top=10;
    Rect.right=600;
    Rect.bottom=380;

    g_pFont->DrawText(NULL, L"Этот текст мы вывели на экран", -1,
             &Rect, DT_CENTER | DT_VCENTER , D3DXCOLOR(1.0,1.0,1.0,1.0));

    g_pSwapChain->Present( 0, 0 );
}
```

Итак, мы обеспечили инициализацию приложения для вывода текста с помощью Direct3D 10 и сам вывод текста на экран (функции `InitDirect3D10()` и `RenderScene()`, соответственно), осталось обеспечить корректное завершение программы, то есть — освобождение ресурсов. Открываем функцию `Cleanup()`, нам нужно добавить всего одну строку — освобождение шрифтового объекта:

```
if( g_pFont ) g_pFont->Release();
```

Обновленная функция `Cleanup()` полностью представлена ниже:

```
void Cleanup()
{
    if( g_pd3dDevice ) g_pd3dDevice->ClearState();

    if( g_pFont ) g_pFont->Release();

    if( g_pRenderTargetView ) g_pRenderTargetView->Release();
    if( g_pSwapChain ) g_pSwapChain->Release();
    if( g_pd3dDevice ) g_pd3dDevice->Release();
}
```

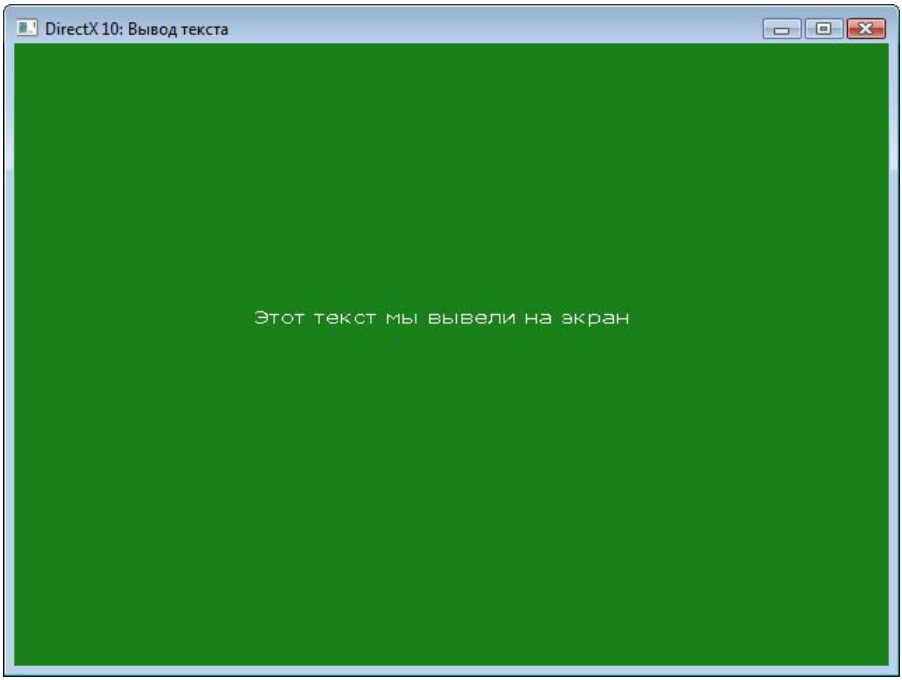

Рис. 4.1. Вывод текста на экран

Ну, вот и все, можно скомпилировать и запустить нашу программу. Результат ее работы можно увидеть на рис. 4.1.

Полный исходный текст программы представлен на листинге 4.1, соответствующий проект можно найти на прилагаемом компакт-диске в директории Glava4\Text. Чтобы освоиться и почувствовать уверенность в своих силах, попробуйте поэкспериментировать с параметрами шрифта и форматирования текста. На следующем этапе нашего пути мы научимся выводить на экран изображение из файла, но для этого нам сначала нужно узнать, что такое ресурсы и как ими пользоваться.

Листинг 4.1

```
//--------------------------------------------------------------------------
//  Вывод текста на экран
//--------------------------------------------------------------------------

#include <windows.h>
#include <d3d10.h>
```

```
#include <d3dx10.h>

// Ширина и высота окна
#define WINDOW_WIDTH  640
#define WINDOW_HEIGHT 480

//--------------------------------------------------------------------------
// Глобальные переменные
//--------------------------------------------------------------------------
HWND        g_hWnd = NULL;
D3D10_DRIVER_TYPE       g_driverType = D3D10_DRIVER_TYPE_NULL;
ID3D10Device*           g_pd3dDevice = NULL;
IDXGISwapChain*         g_pSwapChain = NULL;
ID3D10RenderTargetView* g_pRenderTargetView = NULL;

LPD3DX10FONT            g_pFont = NULL;

//--------------------------------------------------------------------------
// Прототипы функций
//--------------------------------------------------------------------------
HRESULT             InitWindow( HINSTANCE hInstance, int nCmdShow );
HRESULT             InitDirect3D10();
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);
void                Cleanup();
void                RenderScene();

//--------------------------------------------------------------------------
// С этой функции начинается выполнение программы
//--------------------------------------------------------------------------
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow )
{
    // Создаем окно приложения
    if( FAILED( InitWindow( hInstance, nCmdShow ) ) )
        return 0;
// Инициализируем Direct3D
    if( FAILED( InitDirect3D10() ) )
    {
        Cleanup();
```

```
        return 0;
    }

    // Цикл обработки сообщений
    MSG msg = {0};
    while( WM_QUIT != msg.message  )
    {
        if( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) )
        {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
        }
        else
        {
            RenderScene();
        }

    }
    Cleanup();
    return (int) msg.wParam;
}

//--------------------------------------------------------------------------
// Регистрация класса и создание окна
//--------------------------------------------------------------------------
HRESULT InitWindow( HINSTANCE hInstance, int nCmdShow )
{
    // Регистрируем класс окна
    WNDCLASSEX wc;
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style          = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc    = WndProc;
    wc.cbClsExtra     = 0;
    wc.cbWndExtra     = 0;
    wc.hInstance      = hInstance;
    wc.hIcon          = NULL;
    wc.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszMenuName   = NULL;
```

```
    wc.lpszClassName  = L"SimpleWindowClass";
    wc.hIconSm        = LoadIcon(NULL, IDI_APPLICATION);
    if( !RegisterClassEx(&wc) )
        return E_FAIL;

    // Создаем окно
    g_hWnd = CreateWindow(
             L"SimpleWindowClass",
             L"DirectX 10: Вывод текста",
             WS_OVERLAPPEDWINDOW,
             CW_USEDEFAULT,
             CW_USEDEFAULT,
             WINDOW_WIDTH,
             WINDOW_HEIGHT,
             NULL,
             NULL,
             hInstance,
             NULL);

       // Если не удалось создать окно - выходим из функции
    if( !g_hWnd )
        return E_FAIL;
       // Отображаем окно на экране
    ShowWindow( g_hWnd, nCmdShow );
       UpdateWindow(g_hWnd);

    return S_OK;
}

//--------------------------------------------------------------------------
// Инициализация Direct3D
//--------------------------------------------------------------------------
HRESULT InitDirect3D10()
{
       HRESULT hr = S_OK;

    // Узнаем размеры клиентской области окна
    RECT rc;
    GetClientRect( g_hWnd, &rc );
```

```
UINT width = rc.right - rc.left;
UINT height = rc.bottom - rc.top;

   // Список возможных типов устройства
   D3D10_DRIVER_TYPE driverTypes[] =
{
    D3D10_DRIVER_TYPE_HARDWARE,
    D3D10_DRIVER_TYPE_REFERENCE,
};
UINT numDriverTypes = sizeof(driverTypes) / sizeof(driverTypes[0]);

// Заполняем структуру
DXGI_SWAP_CHAIN_DESC sd;
ZeroMemory( &sd, sizeof(sd) );
sd.BufferCount = 1;
sd.BufferDesc.Width = width;
sd.BufferDesc.Height = height;
sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
sd.BufferDesc.RefreshRate.Numerator = 60;
sd.BufferDesc.RefreshRate.Denominator = 1;
sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
sd.OutputWindow = g_hWnd;
sd.SampleDesc.Count = 1;
sd.SampleDesc.Quality = 0;
sd.Windowed = TRUE;

// Пытаемся создать устройство, проходя по списку
// как только получилось - выходим из цикла
for( UINT driverTypeIndex = 0; driverTypeIndex < numDriverTypes;
     driverTypeIndex++ )
{
    g_driverType = driverTypes[driverTypeIndex];
    hr = D3D10CreateDeviceAndSwapChain( NULL, g_driverType,
                  NULL, 0, D3D10_SDK_VERSION, &sd,
                  &g_pSwapChain, &g_pd3dDevice );
    if( SUCCEEDED( hr ) )
        break;
}
if( FAILED(hr) )
```

```
        return hr;

    // Представление данных для
    // буфера визуализации
    ID3D10Texture2D *pBackBuffer;
    // Получим доступ к вторичному буферу с индексом 0
    hr = g_pSwapChain->GetBuffer( 0, __uuidof( ID3D10Texture2D ),
                                  (LPVOID*)&pBackBuffer );
    if( FAILED(hr) )
        return hr;
// Создадим представление данных
    hr = g_pd3dDevice->CreateRenderTargetView( pBackBuffer, NULL,
                                               &g_pRenderTargetView );
    pBackBuffer->Release();
    if( FAILED(hr) )
        return hr;

    g_pd3dDevice->OMSetRenderTargets( 1, &g_pRenderTargetView, NULL );

    // Настроим область отображения
    D3D10_VIEWPORT vp;
    vp.Width = width;
    vp.Height = height;
    vp.MinDepth = 0.0f;
    vp.MaxDepth = 1.0f;
    vp.TopLeftX = 0;
    vp.TopLeftY = 0;
    g_pd3dDevice->RSSetViewports( 1, &vp );

    // Создаем шрифтовой объект
    D3DX10CreateFont( g_pd3dDevice, 14, 8, 1, 1, FALSE, 0, 0, 0,
                      DEFAULT_PITCH|FF_MODERN, L"Verdana", &g_pFont);

    return S_OK;
}

//--------------------------------------------------------------------------
// Прорисовка трехмерной сцены
//--------------------------------------------------------------------------
```

```
void RenderScene()
{
    // Очищаем вторичный буфер
    // (компоненты красного, зеленого, синего, прозрачность)
    float ClearColor[4] = { 0.1f, 0.5f, 0.1f, 1.0f };
    g_pd3dDevice->ClearRenderTargetView( g_pRenderTargetView,
                                         ClearColor );

    //Размеры прямоугольника для форматирования текста
    RECT Rect;
    Rect.left=10;
    Rect.top=10;
    Rect.right=600;
    Rect.bottom=380;

    g_pFont->DrawText(NULL, L"Этот текст мы вывели на экран", -1,
           &Rect, DT_CENTER | DT_VCENTER , D3DXCOLOR(1.0,1.0,1.0,1.0));

    g_pSwapChain->Present( 0, 0 );
}
//--------------------------------------------------------------------------
// Обработка сообщений
//--------------------------------------------------------------------------
LRESULT CALLBACK WndProc( HWND hWnd, UINT message, WPARAM wParam,
                          LPARAM lParam )
{
    switch (message)
    {
        case WM_DESTROY:
            PostQuitMessage(0);
            break;

        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }

    return 0;
}
```

```
//--------------------------------------------------------------------------
// Очищаем память
//--------------------------------------------------------------------------
void Cleanup()
{
    if( g_pd3dDevice ) g_pd3dDevice->ClearState();

    if( g_pFont ) g_pFont->Release();

    if( g_pRenderTargetView ) g_pRenderTargetView->Release();
    if( g_pSwapChain ) g_pSwapChain->Release();
    if( g_pd3dDevice ) g_pd3dDevice->Release();
}
```

Ресурсы в DirectX 10

С чего же начать? Бывают природные ресурсы, их человек использует в своей деятельности для производства каких-то материальных благ. Бывают ресурсы в DirectX 10, их использует программист, тоже человек, для получения нужного ему результата. Эти ресурсы фактически представляют собой области памяти, к которым имеет доступ графический конвейер Direct3D 10 (о нем мы поговорим немного позже, в *гл. 5*). Ресурсы — это своеобразный способ организации доступа к памяти для более эффективного ее использования графической библиотекой. К ресурсам можно отнести входные данные о геометрии, ресурсы шейдеров, текстуры и т. д. Наверное, главной особенностью ресурсов является возможность установить желаемый уровень доступа к ресурсу со стороны центрального процессора и процессора видеокарты. Уровень доступа подразумевает наличие разрешения на чтение и запись. Можно сделать ресурс доступным исключительно для центрального процессора, либо наоборот, только для процессора видеокарты, либо для них обоих. От предоставленного доступа зависит скорость работы графической части программы. Все ресурсы делятся на два основных типа:

- Буферные ресурсы. Буферный ресурс представляет собой набор данных, каждый элемент которого может иметь свой собственный тип, или даже свой собственный формат. Когда буферный ресурс привязывается к графическому конвейеру, приложение предоставляет информацию о том, каким образом читать и интерпретировать данные, которые ресурс содержит.
- Текстурные ресурсы. Текстурный ресурс, в отличие от буферного, представляет собой структурированный набор данных. Данные текстурного

ресурса представляют собой один либо несколько подресурсов (subresource), которые уже, в свою очередь, являются массивами текселей и несут информацию об изображении.

Непонятного, наверное, много. Начнем разбираться. Буферный ресурс — это просто выделенная область памяти, которую приложение может использовать по своему усмотрению. Текстурный ресурс предназначен для хранения информации о текстурах. Тексел (сокращение от "текстурный элемент") — это минимальный элемент текстуры, который может быть считан либо записан. Проще говоря, тексели — это такие текстурные пикселы. Это все нам знакомо. А вот для чего понадобилось размещать внутри ресурса "один либо несколько подресурсов"? Все дело в том, что бывает необходимо иметь одну и ту же текстуру с разным разрешением, например, для использования мипмаппинга. Для хранения этих данных и служат подресурсы. Наглядно организация хранения данных в текстурном ресурсе представлена на рис. 4.2. Именно ресурс этого типа нам потребуется создать, чтобы вывести на экран изображение из файла.

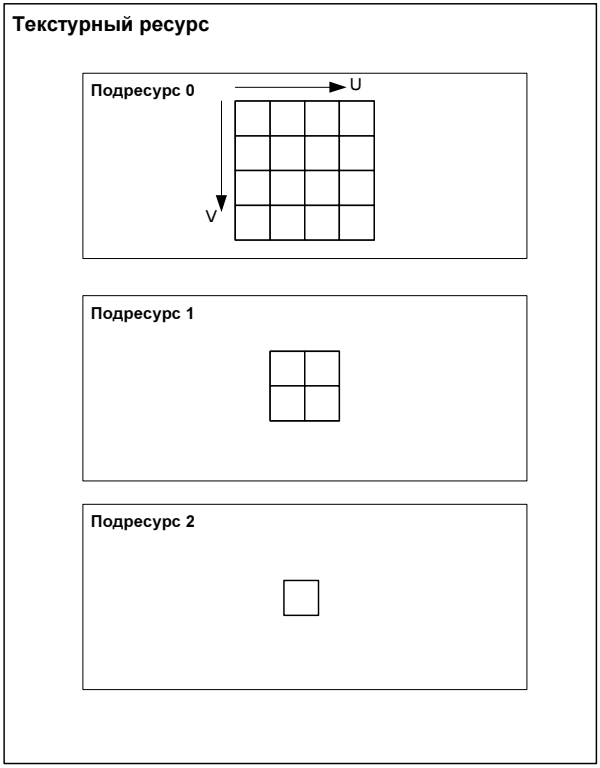

Рис. 4.2. Организация данных внутри текстурного ресурса

Выводим картинку

Как ни странно, в трехмерной графике двухмерные ("плоские") изображения играют далеко не последнюю роль. С их помощью можно, например, нарисовать элементы пользовательского интерфейса или воспользоваться ими для каких-то других целей. В Direct3D 10 для вывода "плоских" изображений (спрайтов) имеется специальный интерфейс `ID3DX10Sprite`. Чтобы организовать вывод спрайтов на экран, необходимо выполнить следующие действия.

1. Создать представление данных как ресурса шейдеров.
2. Создать спрайт, используя имеющееся представление данных.
3. В функции визуализации трехмерной сцены нарисовать спрайт(ы) во вторичный буфер.
4. Освободить ресурсы для интерфейса спрайта и представления данных.

Здесь мы опять сталкиваемся с представлениями данных. Представление данных как ресурса шейдеров используется для всех текстур, которые необходимо связать с графическим конвейером. В случае со спрайтами нам связывать ничего не придется, за нас все сделает сам спрайт.

Напишем программу, которая выводит на экран изображение, содержащееся в файле DirectX10.bmp. Создадим новый проект в Visual Studio и скопируем в него весь исходный текст из минимального приложения Direct3D 10 (см. листинг 2.2). Сейчас будет много новой информации, придется сосредоточиться.

Начнем с глобальных переменных. Чтобы отобразить спрайт, как мы уже знаем, нам понадобятся указатели на интерфейс представления данных ресурса шейдера и собственно на интерфейс спрайта. Добавим их:

```
ID3D10ShaderResourceView* g_pShaderResource = NULL;
ID3DX10Sprite*            g_pSprite = NULL;
```

Теперь обратимся к функции `InitDirect3D10()`, где нам нужно создать представление данных для ресурса шейдера и спрайт. Поскольку мы загружаем данные изображения из файла, можно создать представление данных как ресурса шейдера с помощью функции `D3DX10CreateShaderResourceViewFromFile()`. Рассмотрим прототип этой функции, попутно отметим для себя, какой еще информации для ее вызова нам не хватает.

```
HRESULT D3DX10CreateShaderResourceViewFromFile(
  ID3D10Device *pDevice,
  LPCSTR pSrcFile,
  D3DX10_IMAGE_LOAD_INFO *pLoadInfo,
```

```
  ID3DX10ThreadPump* pPump,
  ID3D10ShaderResourceView** ppShaderResourceView,
  HRESULT* pHResult
);
```

Параметры функции следующие:

- `pDevice` — указатель на интерфейс устройства Direct3D 10;
- `pSrcFile` — указатель на строку с завершающим нулем, в которой содержится имя нужного файла;
- `pLoadInfo` — указатель на структуру типа `D3DX10_IMAGE_LOAD_INFO` с информацией о параметрах загрузки текстуры;
- `pPump` — указатель на интерфейс конвейера обработки потоков, используется, если необходимо асинхронное выполнение команд, мы эту возможность использовать не будем;
- `ppShaderResourceView` — адрес указателя на представление данных ресурса шейдера;
- `pHResult` — указатель на переменную, в которую будет выведен код завершения функции (можно указать `NULL`, а код завершения получать как результат выполнения функции).

Мы можем предоставить все необходимые параметры, кроме заполненной структуры `D3DX10_IMAGE_LOAD_INFO`. Посмотрим, что у этой структуры внутри, ее прототип приведен ниже.

```
typedef struct D3DX10_IMAGE_LOAD_INFO {
  UINT Width;
  UINT Height;
  UINT Depth;
  UINT FirstMipLevel;
  UINT MipLevels;
  D3D10_USAGE Usage;
  UINT BindFlags;
  UINT CpuAccessFlags;
  UINT MiscFlags;
  DXGI_FORMAT Format;
  UINT Filter;
  UINT MipFilter;
  D3DX10_IMAGE_INFO* pSrcInfo;
} D3DX10_IMAGE_LOAD_INFO, *LPD3DX10_IMAGE_LOAD_INFO;
```

В структуре содержатся следующие данные:

- `Width` и `Height` — желаемые ширина и высота текстуры, если фактические размеры больше либо меньше заданного значения, текстура будет отмасштабирована, чтобы соответствовать указанному размеру;
- `Depth` — глубина текстуры, имеет смысл только для объемных текстур;
- `FirstMipLevel` — уровень для мипмаппинга с наибольшим разрешением, в случае, если установлено значение больше 0, после загрузки текстуры уровень `FirstMapLevel` будет назначен на уровень 0;
- `MipLevels` — максимальное количество уровней для мипмаппинга, которое будет иметь текстура;
- `Usage` — режим использования текстурного ресурса;
- `BindFlags` — флаги, определяющие стадии графического конвейера, с которыми будет разрешено связывать текстуру;
- `CpuAccessFlags` — флаги, определяющие, какие разрешения на доступ к ресурсу будет иметь центральный процессор;
- `MiscFlags` — флаги, определяющие прочие, реже используемые свойства ресурса;
- `Format` — формат, который текстура будет иметь после загрузки;
- `Filter` — флаги, задающие фильтрацию текстуры;
- `MipFilter` — флаг, задающий фильтрацию для уровней мипмаппинга;
- `pSrcInfo` — указатель на структуру, в которой содержатся данные о первоначальном изображении (об изображении, которое хранится в файле).

Тут непонятно, наверное, процентов девяносто, но не отчаивайтесь, мы выйдем из положения. Подробно разбирать все значения мы не будем, это перечисление запутает нас еще больше. Будем последовательно заполнять структуру и делать необходимые пояснения. Начнем дополнять функцию `InitDirect3D10()`, вставляя строки после настройки области отображения. В первую очередь объявим структуры данных для информации об изображении из файла (структура типа `D3DX10_IMAGE_INFO`) и для информации о параметрах загрузки текстуры:

```
D3DX10_IMAGE_INFO InfoFromFile;
D3DX10_IMAGE_LOAD_INFO LoadImageInfo;
```

На всякий случай заполним структуры нулями:

```
ZeroMemory( &InfoFromFile, sizeof(InfoFromFile) );
ZeroMemory( &LoadImageInfo, sizeof(LoadImageInfo) );
```

Получим информацию об изображении из файла с помощью функции D3DX10GetImageInfoFromFile():

```
D3DX10GetImageInfoFromFile(L"DirectX10.bmp", NULL, &InfoFromFile, &hr);
```

В качестве параметров этой функции необходимо передать имя файла, указатель на интерфейс обработки потоков (у нас NULL — асинхронный ввод не используется), указатель на структуру, в которую будет выведена информация, и указатель на переменную, куда будет помещен код завершения функции. Воспользуемся полученной из файла информацией при заполнении структуры параметров загрузки текстуры:

```
LoadImageInfo.Width = InfoFromFile.Width;
LoadImageInfo.Height = InfoFromFile.Height;
LoadImageInfo.Depth = InfoFromFile.Depth;
```

Уровень мипмаппинга с наибольшим разрешением, он у нас один-единственный, поэтому указываем 0:

```
LoadImageInfo.FirstMipLevel = 0;
```

Количество уровней мипмаппинга можно скопировать из информации о файле либо, в данном случае, установить в 1:

```
LoadImageInfo.MipLevels = 1;
```

Задаем режим использования ресурса по умолчанию:

```
LoadImageInfo.Usage = D3D10_USAGE_DEFAULT;
```

В следующем поле устанавливаем флаг связывания ресурса с графическим конвейером в качестве ресурса шейдера:

```
LoadImageInfo.BindFlags = D3D10_BIND_SHADER_RESOURCE;
```

Флаги доступа для центрального процессора и флаги прочих свойств зададим равные нулю (центральному процессору доступ к ресурсу не требуется):

```
LoadImageInfo.CpuAccessFlags = 0;
LoadImageInfo.MiscFlags = 0;
```

Формат текстуры возьмем из файла с изображением:

```
LoadImageInfo.Format = InfoFromFile.Format;
```

Флаги, определяющие фильтрацию текстуры, установим в значение "фильтрация отсутствует":

```
LoadImageInfo.Filter = D3DX10_FILTER_NONE;
LoadImageInfo.MipFilter = D3DX10_FILTER_NONE;
```

В последнее поле нам остается поместить адрес переменной, содержащей информацию о первоначальном изображении:

```
LoadImageInfo.pSrcInfo = &InfoFromFile;
```

Когда структура полностью заполнена, вызываем функцию для создания представления данных как ресурса шейдера:

```
hr = D3DX10CreateShaderResourceViewFromFile(g_pd3dDevice,
L"DirectX10.bmp", &LoadImageInfo, NULL, &g_pShaderResource, NULL);
    if( FAILED(hr) )
        return hr;
```

После выполнения функции мы проверяем переменную `hr` и выходим из функции `InitDirect3D10()`, если функция завершилась с ошибкой. В случае успешного завершения мы идем дальше и создаем спрайтовый объект:

```
hr = D3DX10CreateSprite(g_pd3dDevice, 1, &g_pSprite);
    if( FAILED(hr) )
        return hr;
```

Прототип функции `D3DX10CreateSprite()` имеет вид

```
HRESULT D3DX10CreateSprite(
  ID3D10Device *pDevice,
  UINT cDeviceBufferSize,
  LPD3DX10SPRITE *ppSprite
);
```

Параметры функции имеют такие значения:

- `pDevice` — указатель на интерфейс устройства Direct3D 10, которое будет рисовать спрайт;
- `cDeviceBufferSize` — количество отображаемых спрайтов, максимальная величина 4096;
- `ppSprite` — адрес указателя на интерфейс спрайта, этот указатель будет установлен на созданный спрайт.

Этим мы заканчиваем инициализацию Direct3D. Полный исходный текст функции, ее выполняющей, должен выглядеть так:

```
HRESULT InitDirect3D10()
{
    HRESULT hr = S_OK;

    // Размеры клиентской области окна
    RECT rc;
    GetClientRect( g_hWnd, &rc );
    UINT width = rc.right - rc.left;
    UINT height = rc.bottom - rc.top;

    // Список возможных типов устройства
    D3D10_DRIVER_TYPE driverTypes[] =
```

```
{
    D3D10_DRIVER_TYPE_HARDWARE,
    D3D10_DRIVER_TYPE_REFERENCE,
};
UINT numDriverTypes = sizeof(driverTypes) / sizeof(driverTypes[0]);

// Заполняем структуру
DXGI_SWAP_CHAIN_DESC sd;
ZeroMemory( &sd, sizeof(sd) );
sd.BufferCount = 1;
sd.BufferDesc.Width = width;
sd.BufferDesc.Height = height;
sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
sd.BufferDesc.RefreshRate.Numerator = 60;
sd.BufferDesc.RefreshRate.Denominator = 1;
sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
sd.OutputWindow = g_hWnd;
sd.SampleDesc.Count = 1;
sd.SampleDesc.Quality = 0;
sd.Windowed = TRUE;

// Пытаемся создать устройство, проходя по списку,
// как только получилось - выходим из цикла
for( UINT driverTypeIndex = 0; driverTypeIndex < numDriverTypes;
            driverTypeIndex++ )
{
    g_driverType = driverTypes[driverTypeIndex];
    hr = D3D10CreateDeviceAndSwapChain( NULL, g_driverType, NULL, 0,
        D3D10_SDK_VERSION, &sd, &g_pSwapChain, &g_pd3dDevice );
    if( SUCCEEDED( hr ) )
        break;
}
if( FAILED(hr) )
    return hr;

// Представление данных для
// буфера визуализации
ID3D10Texture2D *pBackBuffer;
// Получим доступ к вторичному буферу с индексом 0
```

```
hr = g_pSwapChain->GetBuffer( 0, __uuidof( ID3D10Texture2D ),
                              (LPVOID*)&pBackBuffer );
if( FAILED(hr) )
    return hr;
// Создадим представление данных
hr = g_pd3dDevice->CreateRenderTargetView( pBackBuffer, NULL,
                                           &g_pRenderTargetView );
pBackBuffer->Release();
if( FAILED(hr) )
    return hr;

g_pd3dDevice->OMSetRenderTargets( 1, &g_pRenderTargetView, NULL );

// Настроим область отображения
D3D10_VIEWPORT vp;
vp.Width = width;
vp.Height = height;
vp.MinDepth = 0.0f;
vp.MaxDepth = 1.0f;
vp.TopLeftX = 0;
vp.TopLeftY = 0;
g_pd3dDevice->RSSetViewports( 1, &vp );

D3DX10_IMAGE_INFO InfoFromFile;
D3DX10_IMAGE_LOAD_INFO LoadImageInfo;

ZeroMemory( &InfoFromFile, sizeof(InfoFromFile) );
ZeroMemory( &LoadImageInfo, sizeof(LoadImageInfo) );

// Прочтем информацию об изображении из файла
hr = D3DX10GetImageInfoFromFile( L"DirectX10.bmp", NULL,
                                 &InfoFromFile, NULL );

LoadImageInfo.Width = InfoFromFile.Width;
LoadImageInfo.Height = InfoFromFile.Height;
LoadImageInfo.Depth = InfoFromFile.Depth;
LoadImageInfo.FirstMipLevel = 1;
LoadImageInfo.MipLevels = InfoFromFile.MipLevels;
LoadImageInfo.Usage = D3D10_USAGE_DEFAULT;
```

```
    LoadImageInfo.BindFlags = D3D10_BIND_SHADER_RESOURCE ;
    LoadImageInfo.CpuAccessFlags = 0;
    LoadImageInfo.MiscFlags = 0;
    LoadImageInfo.Format = InfoFromFile.Format;
    LoadImageInfo.Filter = D3DX10_FILTER_NONE;
    LoadImageInfo.MipFilter = D3DX10_FILTER_NONE;
    LoadImageInfo.pSrcInfo = &InfoFromFile;

    hr = D3DX10CreateShaderResourceViewFromFile( g_pd3dDevice,
      L"DirectX10.bmp", &LoadImageInfo, NULL, &g_pShaderResource, NULL );
    if( FAILED(hr) )
        return hr;

    hr = D3DX10CreateSprite(g_pd3dDevice, 1, &g_pSprite);
    if( FAILED(hr) )
        return hr;

    return S_OK;
}
```

Первые два пункта плана организации работы со спрайтами мы выполнили. Чтобы увидеть нашу картинку на экране, необходимо выводить ее во вторичный буфер во время рисования каждого кадра. Переходим к третьему пункту алгоритма, к редактированию функции `RenderScene()`. Перед внесением каких-либо изменений, познакомимся поближе с интерфейсом `ID3DX10Sprite`. Указатель на него содержится в переменной `g_pSprite`. Методы, имеющиеся у интерфейса, представлены в табл. 4.2.

***Таблица 4.2.** Методы интерфейса ID3DX10Sprite*

Метод	Описание
`ID3DX10Sprite::Begin`	Подготавливает устройство Direct3D 10 к рисованию спрайтов
`ID3DX10Sprite ::DrawSpritesBuffered`	Добавляет массив спрайтов к очереди спрайтов на прорисовку. Метод необходимо вызывать между вызовами методов `ID3DX10Sprite::Begin` и `ID3DX10Sprite::End`, также перед вызовом `ID3DX10Sprite::End` нужно вызвать метод `ID3DX10Sprite::Flush`, чтобы отправить всю очередь спрайтов на прорисовку

Таблица 4.2 *(окончание)*

Метод	Описание
`ID3DX10Sprite ::DrawSpritesImmediate`	Производит прорисовку массива спрайтов, в отличие от метода `ID3DX10Sprite::DrawSpritesBuffered` спрайты сразу же отправляются в устройство Direct3D. Метод также необходимо вызывать между вызовами методов `ID3DX10Sprite::Begin` и `ID3DX10Sprite::End`
`ID3DX10Sprite::End`	Вызывается после метода `ID3DX10Sprite::Flush`, возвращает состояние устройства, которое оно имело до вызова метода `ID3DX10Sprite::Begin`
`ID3DX10Sprite::Flush`	Отправляет всю очередь спрайтов на прорисовку устройству Direct3D
`ID3DX10Sprite::GetDevice`	Извлекает указатель на устройство Direct3D, связанное со спрайтовым объектом
`ID3DX10Sprite ::GetProjectionTransform`	Извлекает матрицу проекции, которая применяется ко всем спрайтам
`ID3DX10Sprite ::GetViewTransform`	Извлекает видовое преобразование, применяемое ко всем спрайтам
`ID3DX10Sprite ::SetProjectionTransform`	Задает матрицу проекции для всех спрайтов
`ID3DX10Sprite ::SetViewTransform`	Задает видовое преобразование координат, которое применяется ко всем спрайтам

Попробуем представить примерную последовательность действий, которую нам предстоит запрограммировать.

1. Подготовить необходимые матрицы и установить их для использования спрайтовым объектом (методы `SetProjectionTransform` и `SetViewTransform`).
2. Вызвать метод `ID3DX10Sprite::Begin`.
3. Прорисовать необходимые спрайты (с помощью метода `DrawSpritesBuffered` или `DrawSpritesImmediate`).
4. При необходимости — вызвать метод `Flush` (если использовался метод `DrawSpritesBuffered`).
5. Вызвать метод `End` для окончания вывода спрайтов.

Так все и сделаем. Откроем функцию `RenderScene()` и будем вставлять новые строки после строки, производящей очистку вторичного буфера. Для преобразований нам понадобятся все три матрицы: мировая матрица, матрица вида и матрица проекции. Объявим соответствующие переменные:

```
D3DXMATRIX mWorld;
D3DXMATRIX mView;
D3DXMATRIX mProjection;
```

Пусть наш спрайт отобразится в начале координат, создадим для такого случая матрицу перемещения (мы ее записываем сразу в мировую матрицу, так как других преобразований у нас нет):

```
D3DXMatrixTranslation(&mWorld, 0.0f, 0.0f, 0.0f);
```

Теперь создадим матрицу проекции:

```
D3DXMatrixPerspectiveFovLH( &mProjection, (float)D3DX_PI/4,
1.333f, 0.0f, 1.0f );
```

> ***Замечание***
>
> При рисовании спрайтов обычно используют параллельную проекцию, без коррекции перспективы. Это обусловлено тем, что все спрайты должны находиться на одной плоскости, на плоскости экрана. Здесь мы используем проекцию с коррекцией перспективы, как и для трехмерной сцены, чтобы показать возможности использования спрайтов совместно с трехмерной графикой. В случае необходимости, для использования параллельной проекции можно применить функцию `D3DXMatrixOrthoLH`.

Установим матрицу проекции спрайтового объекта:

```
g_pSprite->SetProjectionTransform(&mProjection);
```

Создадим видовую матрицу:

```
D3DXVECTOR3 vEyePt ( 0.0f, 0.0f,-3.0f );
D3DXVECTOR3 vLookatPt ( 0.0f, 0.0f, 0.0f );
D3DXVECTOR3 vUpVec ( 0.0f, 1.0f, 0.0f );
D3DXMatrixLookAtLH( &mView, &vEyePt, &vLookatPt, &vUpVec );
```

Установим видовую матрицу спрайтового объекта:

```
g_pSprite->SetViewTransform(&mView);
```

Начинаем прорисовку спрайтов, чтобы обозначить ее начало, вызываем метод `ID3DX10Sprite::Begin`:

```
g_pSprite->Begin( D3DX10_SPRITE_SORT_TEXTURE );
```

Единственный параметр метода содержит флаги, которые сообщают, каким образом требуется рисовать спрайты. Мы используем флаг `D3DX10_SPRITE_SORT_TEXTURE`, это означает, что если у нас имеется множество спрайтов и среди них есть спрайты, содержащие одну и ту же текстуру, то они будут

выводиться в одно время, таким образом улучшая быстродействие. Чтобы двигаться дальше, давайте определимся, каким методом мы будем прорисовывать спрайты. Остановимся на методе `ID3DX10Sprite::DrawSpritesBuffered`. Разберем его прототип:

```
HRESULT DrawSpritesBuffered(
  D3DX10_SPRITE *pSprites,
  UINT cSprites
);
```

Для вызова метода нужно указать два параметра:

- `pSprites` — указатель на массив спрайтов для прорисовки;
- `cSprites` — количество спрайтов в массиве `pSprites`.

В нашем случае массив `pSprites` будет содержать только один элемент. Обратите внимание на тип массива — каждый элемент представляет собой структуру типа `D3DX10_SPRITE`. Давайте с ней тоже познакомимся, ведь нам предстоит ее заполнять, пусть даже и для одного элемента. Итак, структура `D3DX10_SPRITE` определяется следующим образом:

```
typedef struct D3DX10_SPRITE {
  D3DXMATRIX matWorld;
  D3DXVECTOR2 TexCoord;
  D3DXVECTOR2 TexSize;
  D3DXCOLOR ColorModulate;
  ID3D10ShaderResourceView *pTexture;
  UINT TextureIndex;
} D3DX10_SPRITE;
```

Что есть что в этой структуре:

- `matWorld` — мировая матрица спрайта, определяет его положение и ориентацию в пространстве мировых координат;
- `TexCoord` — вектор, направленный из верхнего левого угла текстуры к верхнему левому углу спрайта, выраженный в текстурных координатах (фактически, он определяет смещение текстуры относительно верхнего левого угла спрайта);
- `TexSize` — вектор, направленный из верхнего левого угла спрайта в нижний правый угол спрайта, выраженный в текстурных координатах (фактически, он определяет, сколько раз будет повторяться текстура вдоль вертикальной и горизонтальной осей спрайта);
- `ColorModulate` — цвет, на который необходимо умножить цвет пиксела текстуры перед его визуализацией;

- `pTexture` — указатель на представление данных как ресурса шейдера, содержащее текстуру спрайта;
- `TextureIndex` — индекс текстуры, если текстура не является массивом текстур (как в нашем случае), здесь должно содержаться нулевое значение.

Теперь постепенно будем заполнять эту структуру своими параметрами. Сначала объявим переменную нужного типа:

```
D3DX10_SPRITE SpriteToDraw;
```

Установим мировую матрицу спрайта, ее мы уже заполнили:

```
SpriteToDraw.matWorld = mWorld;
```

Определим смещение текстуры относительно верхнего левого угла спрайта. Нам оно не нужно вообще, ставим нули:

```
SpriteToDraw.TexCoord.x = 0.0f;
SpriteToDraw.TexCoord.y = 0.0f;
```

Теперь разберемся с повторяемостью текстуры вдоль вертикальной и горизонтальной осей спрайта. Нам нужна просто картинка из файла, целиком и без повторов. Это значит, что изображение повторяется один раз вдоль вертикальной оси и один раз вдоль горизонтальной:

```
SpriteToDraw.TexSize.x = 1.0f;
SpriteToDraw.TexSize.y = 1.0f;
```

Цвет пикселов будем умножать на составляющие белого цвета, чтобы избежать искажения цветов изображения из файла:

```
SpriteToDraw.ColorModulate = D3DXCOLOR(1.0f,1.0f,1.0f,1.0f);
```

Зададим указатель на текстуру:

```
SpriteToDraw.pTexture = g_pShaderResource;
```

Поскольку наша текстура не является массивом текстур, присваиваем индексу текстуры значение 0:

```
SpriteToDraw.TextureIndex = 0;
```

Готово, заполнили! Сейчас самое время поставить наш "огромный" массив спрайтов в очередь на прорисовку:

```
hr=g_pSprite->DrawSpritesBuffered(&SpriteToDraw, 1);
    if (FAILED (hr) )
        MessageBox(NULL, L"Ошибка при выводе спрайта", L"ОШИБКА", 0);
```

В случае, если метод завершил работу с ошибкой, будет выведено уведомление об этом. Когда все спрайты поставлены в очередь, нужно отправить их устройству Direct3D 10 на прорисовку:

```
g_pSprite->Flush();
```

После этого мы можем со спокойной совестью завершить вывод спрайтов:

```
g_pSprite->End();
```

Теперь наш спрайт уже прорисован во вторичном буфере, следующая строка произведет переключение буферов, и мы увидим в окне нашего приложения картинку из файла DirectX10.bmp:

```
g_pSwapChain->Present( 0, 0 );
```

Полный исходный текст функции `InitDirect3D10()` приведен ниже.

```
void RenderScene()
{
        HRESULT hr;
        // Очищаем вторичный буфер
        //компоненты красного, зеленого, синего, прозрачность
        float ClearColor[4] = { 0.1f, 0.5f, 0.1f, 1.0f };
        g_pd3dDevice->ClearRenderTargetView( g_pRenderTargetView,
                                             ClearColor );

        D3DXMATRIX mWorld;
        D3DXMATRIX mView;
        D3DXMATRIX mProjection;

        D3DXMatrixTranslation( &mWorld, 0.0f, 0.0f, 0.0f);
        D3DXMatrixPerspectiveFovLH( &mProjection, (float)D3DX_PI/4,
                                    1.333f, 0.0f, 1.0f );

        g_pSprite->SetProjectionTransform(&mProjection);

        D3DXVECTOR3 vEyePt    ( 0.0f, 0.0f,-3.0f );
        D3DXVECTOR3 vLookatPt( 0.0f, 0.0f, 0.0f );
        D3DXVECTOR3 vUpVec    ( 0.0f, 1.0f, 0.0f );
        D3DXMatrixLookAtLH( &mView, &vEyePt, &vLookatPt, &vUpVec );

        g_pSprite->SetViewTransform(&mView);

        g_pSprite->Begin( D3DX10_SPRITE_SORT_TEXTURE );

        D3DX10_SPRITE SpriteToDraw;

        SpriteToDraw.matWorld = mWorld;
```

```
        SpriteToDraw.TexCoord.x = 0.0f;
        SpriteToDraw.TexCoord.y = 0.0f;
        SpriteToDraw.TexSize.x = 1.0f;
        SpriteToDraw.TexSize.y = 1.0f;
        SpriteToDraw.ColorModulate = D3DXCOLOR(1.0f,1.0f,1.0f,1.0f);
        SpriteToDraw.pTexture = g_pShaderResource;
        SpriteToDraw.TextureIndex = 0;

        hr=g_pSprite->DrawSpritesBuffered(&SpriteToDraw, 1);
        if (FAILED (hr) )
           MessageBox( NULL, L"Ошибка при выводе спрайта",
                       L"ОШИБКА", 0 );

        g_pSprite->Flush();
        g_pSprite->End();

        g_pSwapChain->Present( 0, 0 );
}
```

Для нормальной работы программы нам еще нужно добавить в функцию `Cleanup()` освобождение ресурсов спрайтового объекта и представления данных как ресурса шейдера:

```
if ( g_pSprite ) g_pSprite->Release();
if ( g_pShaderResource ) g_pShaderResource->Release();
```

Полностью функция должна выглядеть следующим образом:

```
void Cleanup()
{
    if( g_pd3dDevice ) g_pd3dDevice->ClearState();

    if ( g_pSprite ) g_pSprite->Release();
    if ( g_pShaderResource ) g_pShaderResource->Release();

    if( g_pRenderTargetView ) g_pRenderTargetView->Release();
    if( g_pSwapChain ) g_pSwapChain->Release();
    if( g_pd3dDevice ) g_pd3dDevice->Release();
}
```

Составление программы закончено, откомпилируем и запустим ее. Результат работы можно увидеть на рис. 4.3.

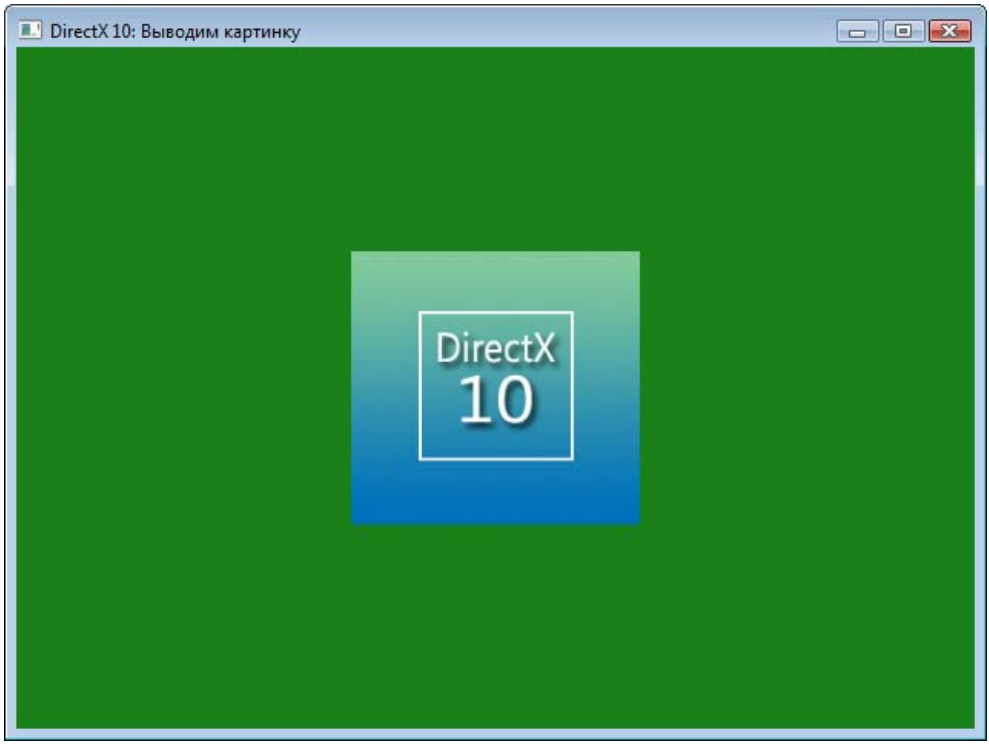

Рис. 4.3. Вывод спрайта на экран

Полный исходный текст представлен в листинге 4.2, соответствующий проект находится на прилагаемом компакт-диске в директории Glava4/Sprite.

Листинг 4.2

```
//-----------------------------------------------------------------------
//  Вывод спрайта на экран
//-----------------------------------------------------------------------
#include <windows.h>
#include <d3d10.h>
#include <d3dx10.h>

// Ширина и высота окна
#define WINDOW_WIDTH  640
#define WINDOW_HEIGHT 480
```

```
//--------------------------------------------------------------------------
// Глобальные переменные
//--------------------------------------------------------------------------
HWND        g_hWnd = NULL;
D3D10_DRIVER_TYPE         g_driverType = D3D10_DRIVER_TYPE_NULL;
ID3D10Device*             g_pd3dDevice = NULL;
IDXGISwapChain*           g_pSwapChain = NULL;
ID3D10RenderTargetView*   g_pRenderTargetView = NULL;
ID3D10ShaderResourceView* g_pShaderResource = NULL;
ID3DX10Sprite*            g_pSprite = NULL;

//--------------------------------------------------------------------------
// Прототипы функций
//--------------------------------------------------------------------------
HRESULT             InitWindow( HINSTANCE hInstance, int nCmdShow );
HRESULT             InitDirect3D10();
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);
void                Cleanup();
void                RenderScene();

//--------------------------------------------------------------------------
// С этой функции начинается выполнение программы
//--------------------------------------------------------------------------
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow )
{
       // Создаем окно приложения
       if( FAILED( InitWindow( hInstance, nCmdShow ) ) )
              return 0;
       // Инициализируем Direct3D
       if( FAILED( InitDirect3D10() ) )
       {
              Cleanup();
              return 0;
       }

       // Цикл обработки сообщений
       MSG msg = {0};
       while( WM_QUIT != msg.message  )
```

```
        {
                if( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) )
                {
                        TranslateMessage( &msg );
                        DispatchMessage( &msg );
                }
                else
                {
                        RenderScene();
                }
        }
        Cleanup();
        return (int) msg.wParam;
}

//--------------------------------------------------------------------------
// Регистрация класса и создание окна
//--------------------------------------------------------------------------
HRESULT InitWindow( HINSTANCE hInstance, int nCmdShow )
{
    // Регистрируем класс окна
    WNDCLASSEX wc;
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style          = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc    = WndProc;
    wc.cbClsExtra     = 0;
    wc.cbWndExtra     = 0;
    wc.hInstance      = hInstance;
    wc.hIcon          = NULL;
    wc.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszMenuName   = NULL;
    wc.lpszClassName  = L"SimpleWindowClass";
    wc.hIconSm        = LoadIcon(NULL, IDI_APPLICATION);
    if( !RegisterClassEx(&wc) )
        return E_FAIL;

    // Создаем окно
    g_hWnd = CreateWindow(
```

```
        L"SimpleWindowClass",
        L"DirectX 10: Выводим картинку",
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        WINDOW_WIDTH,
        WINDOW_HEIGHT,
        NULL,
        NULL,
        hInstance,
        NULL);

    // Если не удалось создать окно - выходим из функции
    if( !g_hWnd )
        return E_FAIL;
    // Отображаем окно на экране
    ShowWindow( g_hWnd, nCmdShow );
    UpdateWindow(g_hWnd);

    return S_OK;
}

//--------------------------------------------------------------------------
// Инициализация Direct3D
//--------------------------------------------------------------------------
HRESULT InitDirect3D10()
{
    HRESULT hr = S_OK;

    // Размеры клиентской области окна
    RECT rc;
    GetClientRect( g_hWnd, &rc );
    UINT width = rc.right - rc.left;
    UINT height = rc.bottom - rc.top;

    // Список возможных типов устройства
    D3D10_DRIVER_TYPE driverTypes[] =
    {
        D3D10_DRIVER_TYPE_HARDWARE,
```

```
    D3D10_DRIVER_TYPE_REFERENCE,
};
UINT numDriverTypes = sizeof(driverTypes) / sizeof(driverTypes[0]);

// Заполняем структуру
DXGI_SWAP_CHAIN_DESC sd;
ZeroMemory( &sd, sizeof(sd) );
sd.BufferCount = 1;
sd.BufferDesc.Width = width;
sd.BufferDesc.Height = height;
sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
sd.BufferDesc.RefreshRate.Numerator = 60;
sd.BufferDesc.RefreshRate.Denominator = 1;
sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
sd.OutputWindow = g_hWnd;
sd.SampleDesc.Count = 1;
sd.SampleDesc.Quality = 0;
sd.Windowed = TRUE;

// Пытаемся создать устройство, проходя по списку,
// как только получилось - выходим из цикла
for( UINT driverTypeIndex = 0; driverTypeIndex < numDriverTypes;
                        driverTypeIndex++ )
{
    g_driverType = driverTypes[driverTypeIndex];
    hr = D3D10CreateDeviceAndSwapChain( NULL, g_driverType, NULL, 0,
              D3D10_SDK_VERSION, &sd, &g_pSwapChain, &g_pd3dDevice );
    if( SUCCEEDED( hr ) )
        break;
}
if( FAILED(hr) )
    return hr;

// Представление данных для
// буфера визуализации
ID3D10Texture2D *pBackBuffer;
// Получим доступ к вторичному буферу с индексом 0
hr = g_pSwapChain->GetBuffer( 0, __uuidof( ID3D10Texture2D ),
                                (LPVOID*)&pBackBuffer );
```

```
if( FAILED(hr) )
    return hr;
// Создадим представление данных
hr = g_pd3dDevice->CreateRenderTargetView( pBackBuffer, NULL,
                                           &g_pRenderTargetView );
pBackBuffer->Release();
if( FAILED(hr) )
    return hr;

g_pd3dDevice->OMSetRenderTargets( 1, &g_pRenderTargetView, NULL );

// Настроим область отображения
D3D10_VIEWPORT vp;
vp.Width = width;
vp.Height = height;
vp.MinDepth = 0.0f;
vp.MaxDepth = 1.0f;
vp.TopLeftX = 0;
vp.TopLeftY = 0;
g_pd3dDevice->RSSetViewports( 1, &vp );

D3DX10_IMAGE_INFO InfoFromFile;
D3DX10_IMAGE_LOAD_INFO LoadImageInfo;

ZeroMemory( &InfoFromFile, sizeof(InfoFromFile) );
ZeroMemory( &LoadImageInfo, sizeof(LoadImageInfo) );

// Читаем информацию об изображении из файла
hr = D3DX10GetImageInfoFromFile( L"DirectX10.bmp", NULL,
                                 &InfoFromFile, NULL );

LoadImageInfo.Width = InfoFromFile.Width;
LoadImageInfo.Height = InfoFromFile.Height;
LoadImageInfo.Depth = InfoFromFile.Depth;
LoadImageInfo.FirstMipLevel = 1;
LoadImageInfo.MipLevels = InfoFromFile.MipLevels;
LoadImageInfo.Usage = D3D10_USAGE_DEFAULT;
LoadImageInfo.BindFlags = D3D10_BIND_SHADER_RESOURCE ;
LoadImageInfo.CpuAccessFlags = 0;
```

```
    LoadImageInfo.MiscFlags = 0;
    LoadImageInfo.Format = InfoFromFile.Format;
    LoadImageInfo.Filter = D3DX10_FILTER_NONE;
    LoadImageInfo.MipFilter = D3DX10_FILTER_NONE;
    LoadImageInfo.pSrcInfo = &InfoFromFile;

    hr = D3DX10CreateShaderResourceViewFromFile(g_pd3dDevice,
                L"DirectX10.bmp", &LoadImageInfo, NULL,
                &g_pShaderResource, NULL);
    if( FAILED(hr) )
        return hr;

    hr = D3DX10CreateSprite(g_pd3dDevice, 1, &g_pSprite);
    if( FAILED(hr) )
        return hr;

    return S_OK;
}

//--------------------------------------------------------------------------
// Прорисовка трехмерной сцены
//--------------------------------------------------------------------------
void RenderScene()
{
    HRESULT hr;
    // Очищаем вторичный буфер
     //компоненты красного, зеленого, синего, прозрачность
    float ClearColor[4] = { 0.1f, 0.5f, 0.1f, 1.0f };
    g_pd3dDevice->ClearRenderTargetView( g_pRenderTargetView,
                                         ClearColor );

    D3DXMATRIX mWorld;
    D3DXMATRIX mView;
    D3DXMATRIX mProjection;

    D3DXMatrixTranslation(&mWorld, 0.0f, 0.0f, 0.0f);
    D3DXMatrixPerspectiveFovLH( &mProjection, (float)D3DX_PI/4, 1.333f,
                                0.0f, 1.0f );
```

```
    g_pSprite->SetProjectionTransform(&mProjection);

    D3DXVECTOR3 vEyePt    ( 0.0f, 0.0f,-3.0f );
    D3DXVECTOR3 vLookatPt( 0.0f, 0.0f, 0.0f );
    D3DXVECTOR3 vUpVec    ( 0.0f, 1.0f, 0.0f );
    D3DXMatrixLookAtLH( &mView, &vEyePt, &vLookatPt, &vUpVec );

    g_pSprite->SetViewTransform(&mView);

    g_pSprite->Begin( D3DX10_SPRITE_SORT_TEXTURE );

    D3DX10_SPRITE SpriteToDraw;

    SpriteToDraw.matWorld = mWorld;

    SpriteToDraw.TexCoord.x = 0.0f;
    SpriteToDraw.TexCoord.y = 0.0f;

    SpriteToDraw.TexSize.x = 1.0f;
    SpriteToDraw.TexSize.y = 1.0f;
    SpriteToDraw.ColorModulate = D3DXCOLOR(1.0f,1.0f,1.0f,1.0f);
    SpriteToDraw.pTexture = g_pShaderResource;
    SpriteToDraw.TextureIndex = 0;

    hr=g_pSprite->DrawSpritesBuffered(&SpriteToDraw, 1);
    if (FAILED (hr) )
        MessageBox (NULL,L"Ошибка при выводе спрайта",L"ОШИБКА",0);

    g_pSprite->Flush();
    g_pSprite->End();

    g_pSwapChain->Present( 0, 0 );
}
//--------------------------------------------------------------------------
// Обработка сообщений
//--------------------------------------------------------------------------
LRESULT CALLBACK WndProc( HWND hWnd, UINT message, WPARAM wParam,
LPARAM lParam )
{
```

```
    switch (message)
    {

    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }

    return 0;
}

//--------------------------------------------------------------------------
// Очистка памяти
//--------------------------------------------------------------------------
void Cleanup()
{
    if( g_pd3dDevice ) g_pd3dDevice->ClearState();

    if ( g_pSprite ) g_pSprite->Release();
    if ( g_pShaderResource ) g_pShaderResource->Release();

    if( g_pRenderTargetView ) g_pRenderTargetView->Release();
    if( g_pSwapChain ) g_pSwapChain->Release();
    if( g_pd3dDevice ) g_pd3dDevice->Release();
}
```

В этой главе мы с вами научились выводить на экран текст и картинки из графических файлов. Пока мы еще даже не прикоснулись к трехмерной графике, от нее нас отделяет последняя ступенька. Давайте шагнем на нее.

Глава 5

Слово о шейдерах

Что такое шейдеры?

Мы уже научились выводить на экран двухмерные изображения. Чтобы успешно двигаться дальше, нам необходимо знать хотя бы основы программирования шейдеров. Шейдеры появились в восьмой версии DirectX. Поначалу количество команд на шейдер было ограничено, и писать их нужно было на ассемблере. То есть для того, чтобы написать шейдер, нужно было выучить еще один язык — ассемблер для работы с регистрами процессора видеокарты. Текст шейдера записывался в текстовый файл и выглядел примерно так:

```
; Фрагмент вершинного шейдера
; на ассемблере
vs_1_1;
dcl_position v0 ;
dcl_texcoord v4;
mul r1, v0, c2;
add oPos, r1, c4;
mov oT0.xy, v4;
mov oD0, c4;
```

Выглядит несколько загадочно, не правда ли? Программирование на ассемблере, с одной стороны, давало программисту полный контроль над использованием команд и регистров процессора видеокарты, но с другой — тексты на ассемблере труднее для понимания, особенно если они написаны другим человеком, он сложнее в освоении.

Шейдер в Direct3D 10 — это, конечно, тоже программа, но написанная на языке HLSL (High Level Shading Language — язык шейдеров высокого уровня), которая выполняется процессором видеокарты и служит для получения различных визуальных эффектов. Под визуальными эффектами нужно понимать

и получение простой однотонной закраски, и сложный расчет освещения от нескольких источников света. В общем, если мы хотим научиться писать программы под Direct3D 10, без шейдеров нам не обойтись.

Для начала познакомимся с графическим конвейером в Direct3D (рис. 5.1).

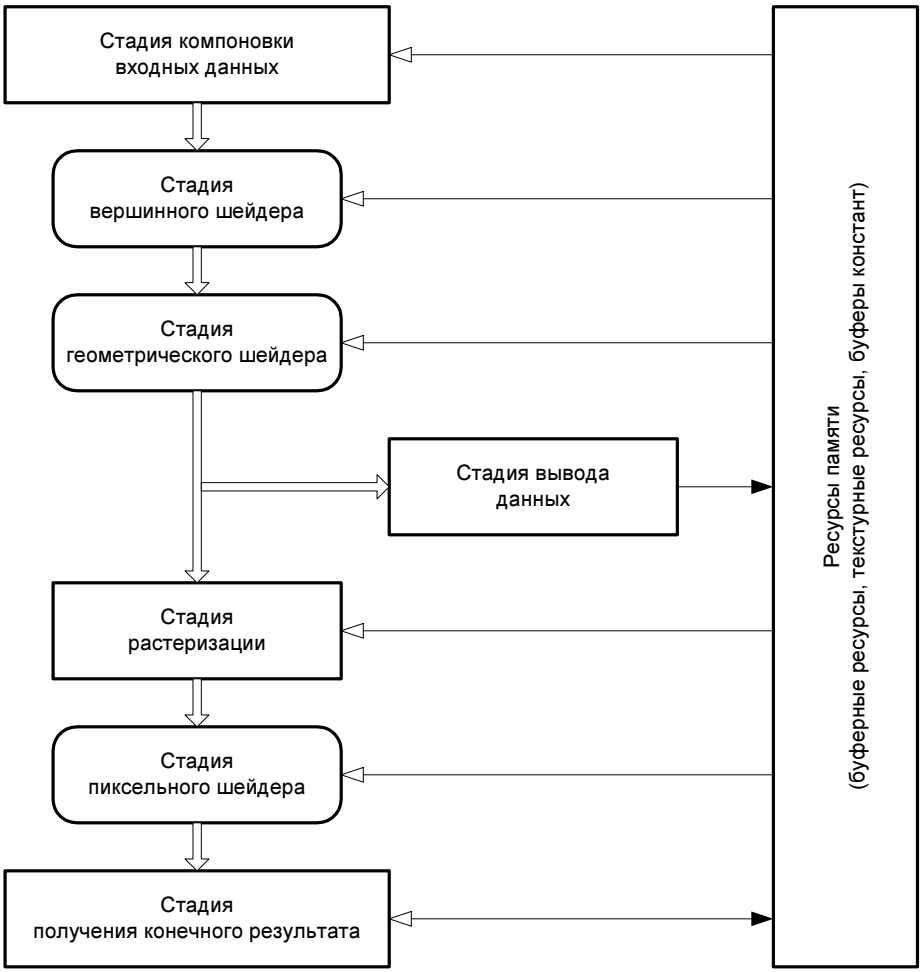

Рис. 5.1. Графический конвейер Direct3D 10

При отображении графики данные проходят через следующие стадии.

- Стадия компоновки входных данных. На этой стадии происходит подготовка данных о примитивах (треугольниках, отрезках и вершинах) и передача их для обработки на последующих этапах конвейера.

- Стадия вершинного шейдера. Как ясно из названия, вершинный шейдер производит обработку вершин. В основном, он выполняет такие операции, как геометрические преобразования, и расчет освещенности грани на основе освещенности вершин. Вершинный шейдер всегда принимает данные одной вершины и возвращает тоже данные только одной результирующей вершины.
- Стадия геометрического шейдера. Геометрический шейдер обрабатывает примитив целиком. На вход подаются геометрические данные примитива (три вершины для треугольника, две вершины для отрезка, и одна — для точки). Дополнительно каждый примитив может иметь информацию о смежных примитивах: три дополнительные вершины для треугольника и две дополнительные вершины для отрезка. Геометрический шейдер также может в некоторых пределах осуществлять расширение и сокращение геометрических размеров. При получении одного примитива на входе, геометрический шейдер может либо проигнорировать его, либо породить один и более новых примитивов. То есть с помощью геометрического шейдера можно создавать новые вершины прямо "на лету".
- Стадия вывода данных. На этой стадии можно передать данные о примитивах из конвейера в оперативную память компьютера (все помнят, что шейдеры выполняются процессором видеокарты?). Данные можно выгрузить в память, после чего передать их на растеризацию либо не передавать. Выгруженные данные можно снова передать на вход графического конвейера либо каким-то образом манипулировать ими с помощью центрального процессора.
- Стадия растеризации. На стадии растеризации происходит отсечение примитивов (не попадающих в область видимости частей) и подготовка примитивов к работе пиксельного шейдера.
- Стадия пиксельного шейдера. Выполняет такие операции, как смешивание текстур (texture blending), расчет модели освещения, попиксельный расчет нормалей и нанесение отражения окружающего мира (environmental mapping). Другими словами, пиксельный шейдер получает на входе данные о примитивах, преобразованных в пикселы, и в процессе своей работы устанавливает цвет каждого пиксела индивидуально.
- Стадия получения конечного результата. Данная стадия отвечает за объединение выходных данных различного рода (данных от пиксельного шейдера, информация из буфера глубины (depth buffer) и буфера шаблона (stencil buffer)) с содержимым буфера визуализации (render target) для создания итогового результата работы конвейера.

Как можно заключить из приведенной схемы, результат работы графического конвейера Direct3D 10 — это результат работы трех видов шейдеров. Теперь

самое время познакомиться с основами языка для написания шейдеров и попробовать свои силы в написании простейшего шейдера, который мы будем использовать уже в следующей главе при создании программы, выводящей на экран треугольник. Мы, конечно же, не сможем охватить полностью весь язык, такое описание само по себе достойно отдельной книги, но мы постараемся приобрести знания для понимания структуры программ-шейдеров и создания таких программ для наших учебных целей. Прежде чем мы перейдем к рассмотрению языка, давайте во избежание путаницы договоримся, что приложением или программой будем называть основную программу (на языке C++), а программу на языке HLSL — всегда будем называть "шейдер". Итак…

Язык HLSL

Изучение HLSL не будет трудным для человека, знающего язык C. Язык для написания шейдеров унаследовал от него очень многое, тем не менее, есть ряд особенностей, на которые нужно обратить внимание.

Текст программы на HLSL записывается в обычный текстовый файл с помощью любого подходящего текстового редактора (например — стандартный "Блокнот" в Windows). Компиляция шейдера может осуществляться как непосредственно при работе программы, так и на стадии ее проектирования. Структура программы-шейдера выглядит примерно следующим образом.

```
// Глобальная переменная верхнего уровня
float GlobalVar;

// функция верхнего уровня
void function(
in float4 position: POSITION0 // аргумент верхнего уровня
              )
{
  float LocalVar; // локальная переменная
  function2(...)
}

void function2() // какая-то другая функция
{
  ...
}
```

Как можно заметить, глобальные переменные верхнего уровня, так же, как это принято в языке C, объявляются вне функций. Функция верхнего уровня — любая функция, которая вызывается приложением, в отличие от тех, которые вызываются другими функциями шейдера.

Комбинировать шейдеры вместе можно с помощью файла эффектов. Структура файла эффектов представлена на рис. 5.2.

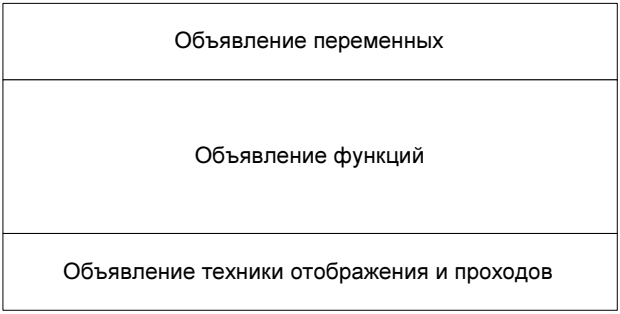

Рис. 5.2. Структура файла эффектов

Собственно в этом файле содержатся переменные и функции шейдеров, а после них вводятся описания техник отображения. Здесь конкретно указывается, какой из имеющихся шейдеров назначить к выполнению при использовании той или иной техники. Например, можно сделать две разные техники отображения, которые будут обращаться к одному и тому же вершинному шейдеру, но будут использовать разные пиксельные шейдеры, в зависимости от выбранных пользователем установок качества расчета освещения.

Мы описали шейдеры и файлы эффектов в общих чертах, продолжим знакомство с языком HLSL и посмотрим, что собой представляют…

Типы данных и объявление переменных

Традиционно рассмотрение языка программирования начинают с описания поддерживаемых типов данных и способа описания переменных. Начнем с самого простого.

Базовые типы

Основных, базовых типов всего шесть:

- `bool` — булевский тип, значения либо "истина", либо "ложь";
- `int` — 32-битное целое число со знаком;

- `uint` — 32-битное целое число без знака;
- `half` — 16-битное число с плавающей точкой, нужно иметь в виду, что этот тип присутствует лишь для совместимости, фактически для хранения данных такого типа все равно используется тип `float`;
- `float` — 32-битное число с плавающей точкой;
- `double` — число с двойной точностью, т. е. 64-битное число с плавающей точкой.

Объявление переменных осуществляется так же, как это делается в языках C/C++: сначала указывается тип, затем имя переменной, например:

```
bool bLogic; // булева (логическая) переменная
int iVar1; // переменная целого типа со знаком
float fVar2; // число с плавающей точкой
double dVar3; //число с двойной точностью
```

Вместе с объявлением переменной можно одновременно присваивать ей значение (инициализировать ее), например:

```
bool bLogic = true;
int iVar1 = 3;
float fVar2 = 4.0f;
double dVar3 = 5.9;
```

Пока что, как будто, ничего нового не появилось, все это нам очень хорошо знакомо. Типы переменных, которые мы рассмотрим сейчас, нам еще не встречались, но без них, как мы уже знаем, программирование трехмерной графики превращается в довольно трудоемкое и малоприятное занятие. Речь, конечно же, идет о векторах и матрицах.

Векторы

Как переменная, вектор фактически представляет собой одномерный массив, в котором содержится до четырех элементов одного типа (любого из базовых). Объявление выглядит таким образом: указывается тип вектора, количество элементов и имя переменной. Например:

```
int2 iVector2; // Вектор, содержащий два целых элемента
int3 iVector3; // Три целых элемента
float4 fVector; // Четыре элемента с плавающей точкой
double1 dVector; // Один элемент двойной точности
```

Инициализация вектора осуществляется аналогично инициализации обычных переменных:

```
int2 iVector2 = {1, 2};
int3 iVector3 = {1, 2, 3};
```

```
float4 fVector = {0.1f, 0.2f, 0.3f, 0.4f} ;
double1 dVector = 1.123;
```

Для доступа к компонентам вектора имеется два способа именования (назначения имен) элементов по порядку следования:

1. Для информации о координатах — `x`, `y`, `z`, `w`.
2. Для информации о цвете — `r`, `g`, `b`, `a`.

Пользоваться можно любым из них. Именование элементов позволяет сделать исходный текст более понятным при работе с векторами, содержащими данные одного типа, но разной природы. Например:

```
// Тестовая переменная:
float fTest=0.0f;
// Вектор из четырех элементов:
float4 fVector = {0.1f, 0.2f, 0.3f, 0.4f};
// Предположим, что переменная fVector содержит координаты
fTest = fVector.x; // fTest приняло значение 0.1
fTest = fVector.y; // fTest приняло значение 0.2
fTest = fVector.z; // fTest приняло значение 0.3
fTest = fVector.w; // fTest приняло значение 0.4
// А сейчас — что информацию о цвете
fTest = fVector.g; // fTest приняло значение 0.2
fTest = fVector.a; // fTest приняло значение 0.4
```

Можно присваивать значение сразу нескольким элементам, но смешивать способы именования не допускается, например:

```
// Вектор из двух элементов
float2 temp;
temp = fVector.xy;// temp содержит {0.1, 0.2}
temp = fVector.ra;// temp содержит {0.1, 0.4}

temp = fVector.xg;// НЕДОПУСТИМО: смешение способов именования
```

Так выглядит работа с векторами, т. е. с одномерными переменными. Согласитесь, все реализовано довольно просто. Пойдем дальше и посмотрим на их "старших братьев", двухмерные переменные или, проще говоря, …

Матрицы

Как переменная, матрица представляет собой двухмерный массив, состоящий из элементов одного из базовых типов. Тип всех элементов должен быть одинаковым.

Объявление матрицы производится так: указывается тип элементов матрицы, сразу после него — размерность (количество строк на количество столбцов), и имя переменной. Максимальное количество столбцов или строк матрицы — четыре. Примеры объявления матриц:

```
// Целочисленная матрица из двух строк и двух столбцов
int2x2 iMatrix;
// Целочисленная матрица из четырех строк и двух столбцов
int4x2 iMatrix2;
// Целочисленная матрица из двух строк и четырех столбцов
int2x4 iMatrix3;
// Целочисленная матрица из четырех строк и четырех столбцов
int4x4 iMatrix4;
// Матрица из чисел с плавающей точкой,
// две строки и четыре столбца
float2x4 fMatrix5;
// Матрица из чисел с двойной точностью,
// четыре строки и четыре столбца
double4x4 dMatrix6;
```

Инициализация матриц осуществляется следующим образом:

```
float2x4 fMatrix = {1.1f, 1.2f, 1.3f, 1.4f ,// 1-я строка
                    2.1f, 2.2f, 2.3f, 2.4f  // 2-я строка
                   };
```

Для доступа к элементам матрицы существует два способа именования элементов: с отсчетом индексов от нуля и от единицы. При отсчете индексов от нуля элементы матрицы обозначаются следующим образом:

```
_m00, _m01, _m02, _m03
_m10, _m11, _m12, _m13
_m20, _m21, _m22, _m23
_m30, _m31, _m32, _m33
```

При отсчете индексов от единицы элементы матрицы обозначаются так:

```
_11, _12, _13, _14
_21, _22, _23, _24
_31, _32, _33, _34
_41, _42, _43, _44
```

Рассмотрим пример чтения элементов матрицы:

```
// Тестовая переменная:
float fTest=0.0f;
```

```
// Матрица
float2x2 fMatrix = {1.1f, 1.2f,
                    2.1f, 2.2f};
// Отсчет индексов от нуля
fTest = fMatrix._m00; // fTest приняло значение 1.1
fTest = fMatrix._m11; // fTest приняло значение 2.2
// Отсчет индексов от единицы
fTest = fMatrix._12; // fTest приняло значение 1.2
fTest = fMatrix._21; // fTest приняло значение 2.1
```

Так же, как в случае с векторами, можно считывать значения нескольких элементов за одно обращение и так же, как и при работе с векторами, нельзя смешивать способы именования. Примеры:

```
// Вектор из двух элементов
float2 temp;
// Матрица
float2x2 fMatrix = {1.1f, 1.2f,
                    2.1f, 2.2f};
temp = fMatrix._m00_m01; // temp содержит {1.1, 1.2}
temp = fMatrix._22_11; // temp содержит {2.2, 1.1}


temp = fMatrix._m01_12; // НЕДОПУСТИМО: смешение способов именования
```

> ***Замечание***
>
> Еще раз обратите внимание, что порядок указания индексов элемента матрицы отличается от привычного порядка следования координат. У элемента матрицы первым идет индекс, указывающий на строку, в которой находится элемент, затем — на столбец. То есть здесь сначала указывается "координата Y", а затем "координата X".

Математические действия с векторами и матрицами

И с векторами, и с матрицами нам нужно будет выполнять те или иные математические операции. Для того чтобы получать программы с предсказуемым результатом вычислений, важно понять одну особенность действий такого рода переменными: в HLSL эти действия осуществляются поэлементно. О чем идет речь? Пусть у нас есть два вектора: `v1` и `v2`, из четырех элементов каждый и мы выполняем следующую строку программы:

```
float4 ResVect = v1*v2;
```

Результат работы этого фрагмента будет равносилен результату выполнения следующих строк:

```
ResVect.x = v1.x*v2.x;
ResVect.y = v1.y*v2.y;
```

```
ResVect.z = v1.z*v2.z;
ResVect.w = v1.w*v2.w;
```

Обратите внимание, что это именно поэлементное умножение векторов, а не их скалярное произведение, в результате которого получается одно число. Вспомним, как вычислить скалярное произведение:

```
float DotProduct = v1.x*v2.x + v1.y*v2.y + v1.z*v2.z + v1.w*v2.w;
```

Скалярное произведение векторов можно получить, используя встроенную функцию `dot()` языка HLSL.

Действия с матрицами также осуществляются поэлементно. И ошибиться здесь еще проще. Тип матриц в библиотеке D3DX реализован таким образом, что результат умножения двух матриц можно получить так же, как и результат перемножения любых других переменных. В языке HLSL все обстоит несколько иначе:

```
float2x2 mat1, mat2;
float2x2 mat3 = mat1*mat2;
```

Умножение будет выполнено поэлементно, как в предыдущем примере с векторами. В частности, первый элемент матрицы `mat3` будет иметь вид

```
mat3._m00 = mat1._m00*mat2._m00;
```

Напомню, что при матричном перемножении двух матриц размерностью 2×2 элемент `_m00` результирующей матрицы должен определяться следующим выражением:

```
mat3._m00 = mat1._m00*mat2._m00+
            mat1._m01*mat2._m10;
```

Перемножить две матрицы можно с использованием встроенной функции `mul()`. С этой функцией, а также с некоторыми другими встроенными функциями языка HLSL мы еще не раз встретимся на страницах этой книги.

Ну вот, мы и познакомились с основными типами, необходимыми для написания шейдеров на языке HLSL, и даже узнали имена некоторых полезных функций. Это, безусловно, уже немало, но еще недостаточно для составления нашего первого шейдера. Чтобы приступить к его написанию, важно представлять, как обрабатываются данные, поступающие в шейдер.

Входные данные шейдеров. Семантика

Данные, которые поступают в вершинные и пиксельные шейдеры, можно разделить на два вида: постоянные и переменные. Переменные данные — это те, значение которых уникально для каждого выполнения шейдера, то есть при каждом запуске шейдера они разные. Постоянные, как следует из назва-

ния, — такие, которые при многократных выполнениях шейдера сохраняют свои значения.

Применительно к вершинному шейдеру переменными входными данными будут являться положение вершин в пространстве, информация о нормалях, текстурных координатах и т. д. К постоянным данным можно отнести цвет материала, мировую матрицу, матрицу вида и матрицу проекции, так как они применяются ко всем вершинам одинаково.

При составлении программ нужно указывать, к какому виду относятся получаемые входные данные, эта информация принимается в расчет при компиляции шейдера (делается соответствующее распределение регистров видеокарты). Постоянными входными данными считаются все глобальные переменные, объявленные вне функций, на самом "верху". Так что наиболее простой способ указать на постоянные данные, — это держать их в глобальных переменных. Другим способом указания служит добавление ключевого слова `uniform` перед нужным аргументом в списке аргументов функции верхнего уровня.

Вообще, параметры функций верхнего уровня обязательно должны быть помечены либо как неизменные (с помощью ключевого слова `uniform`), либо, если параметр относится к переменным входным данным, с помощью семантических ключевых слов. В противном случае шейдер скомпилировать не удастся.

Что такое семантика и о каких ключевых словах идет речь? Семантика, применительно к программированию, — это информация, которая помогает связывать выходные данные одних стадий конвейера с соответствующими им входными данными последующих стадий, проще говоря — передавать данные между различными стадиями. И совсем уж попросту говоря, — семантика указывает на назначение параметра, для чего его нужно использовать. Например, вершинный шейдер в выходных данных выдает переменную, содержащую координаты положения вершины в пространстве, и переменную, содержащую информацию о текстурных координатах. После него проходят стадии геометрического шейдера и растеризации, далее начинается стадия пиксельного шейдера. В функции пиксельного шейдера в списке аргументов значатся два параметра. Программисту понятно, что, например, первый параметр должен принимать данные о координатах вершины, а второй — данные о текстурных координатах. Однако как объяснить компьютеру, что он должен распределить данные с предыдущих этапов именно таким образом? Для этого и служат ключевые слова семантики. Они добавляются к выходным параметрам функции вершинного шейдера и к аргументам функции пиксельного шейдера, в результате чего выходные значения пере-

даются параметрам функции с одинаковыми ключевыми словами, с одинаковой семантикой. Все это звучит немного непривычно, но все неясности должны исчезнуть после знакомства с примером шейдера. Но перед этим нам еще нужно рассмотреть…

Объявление функций

Функции в языке HLSL, как и в C, служат для того, чтобы разбить одну большую задачу на несколько меньших задач. Такое разбиение упрощает отладку программы и делает ее текст более ясным для понимания. Функция состоит из описания и следующего за ним тела функции. Вот пример объявления функции:

```
float4 VertexShaderExample(float4 Pos : POSITION ) : POSITION
{
    return mul( Pos, WorldViewProj );
};
```

Тело функции, как и в программах на языке C, находится между открывающей и закрывающей фигурными скобками. Описание же функции включает следующее (по порядку слева направо):

- тип возвращаемого значения (в нашем примере `float4`);
- имя функции (`VertexShaderExample`);
- список аргументов (необязательно), здесь он имеется;
- семантику выходных данных (необязательно), у нас это `POSITION`;
- аннотирование (необязательно), в нашем случае оно не используется.

Обратите внимание на ключевые слова семантики (`POSITION`), они означают, что функция принимает значения, представляющие координаты вершины, и возвращает значения такого же вида в выходном параметре (типа `float4`). Возвращаемое значение функции, как и в языке C, записывается после оператора `return`. Однако далеко не всегда можно обойтись единственным выходным параметром. Организовать вывод нескольких значений нам помогут…

Структуры данных

Структуры объявляются и используются таким же образом, как и в языке C. Объявим структуру, содержащую целое число и вектор из четырех элементов с плавающей точкой:

```
struct StructExample
{
```

```
    int a;
    float4 Vector01;
};
```

Никаких отличий от C! Если структура представляет собой набор входных параметров, то ее элементы также необходимо дополнять семантикой. Попробуем объявить структуру для параметров вершинного шейдера. Будем считать, что шейдер принимает от предыдущей стадии конвейера координаты положения вершины в пространстве, а также текстурные координаты. Структура будет выглядеть примерно так:

```
struct VS_INPUT
{
    float4 Pos : POSITION;
    float4 TexCoord : TEXCOORD;
};
```

Без объявления функций, конечно, программу написать нельзя. Но необходимо также уметь пользоваться готовыми, встроенными функциями, которые предлагает нам язык HLSL.

Встроенные функции HLSL

Язык HLSL предоставляет богатый набор функций для выполнения различных операций, которые могут потребоваться при программировании шейдеров: тригонометрические функции, функции для выполнения действий с матрицами и векторами, функции для работы с текстурами и т. д. Все эти функции мы рассмотреть не сможем, но познакомимся с наиболее полезными для нас, которыми мы будем часто пользоваться. Итак, начнем.

mul (a, b)

Возвращает результат матричного перемножения параметров `a` и `b`. Параметры `a` и `b` могут быть скалярного, векторного и матричного типа. В наших примерах чаще всего мы будем использовать эту функцию для умножения вектора, представляющего собой набор координат вершины, на матрицу преобразования. Пример использования:

```
output.Pos = mul( input.Pos, WorldViewProj );
```

dot (a, b)

Возвращает скалярное произведение векторов `a` и `b`. Вектора должны иметь одинаковую размерность и тип либо `float`, либо `int`. Эта функция впоследствии поможет нам при расчете освещения. Пример использования:

```
float DotProduct = dot( vector1, vector2 );
```

cross (a, b)

Возвращает результат векторного умножения векторов `a` и `b`. Каждый из векторов должен быть типа `float3`. С помощью векторного произведения мы сможем получить вектор нормали к плоскости, в которой лежат вектора `a` и `b`. Пример использования:

```
float3 CrossProduct = cross( vector1, vector2 );
```

distance (a, b)

Возвращает расстояние между точками `a` и `b`. Параметры функции должны представлять собой вектора типа `float`, возвращаемое значение также имеет тип `float`. Пример использования:

```
float distanceAB = distance( point1, point2 );
```

length (v)

Возвращает модуль (длину) вектора `v`. Пример использования:

```
Float VectorLength = length (vector1);
```

normalize (v)

Устанавливает единичную длину (выполняет нормализацию) вектора `v`, сохраняя его направление. Вектор должен быть типа `float` и иметь ненулевую длину: при нулевой длине вектора результат функции не определен. Пример использования:

```
float3 VectorN = {5.0f, 0.0f, 0.0f};
normalize( VectorN );
// сейчас VectorN содержит: {1.0f, 0.0f, 0.0f};
```

saturate (x)

Приводит параметр `x` к интервалу [0,1]:

- если `x` < 0, функция возвращает 0;
- если `x` > 1, функция возвращает 1;
- в любом другом случае функция возвращает значение параметра `x`.

Параметр `x` может представлять собой скаляр, вектор или матрицу типа `float`. Пример использования:

```
float test = 1.123f;
test = saturate( test );
// test = 1.0;
```

cos (x)

Возвращает косинус угла x, выраженного в радианах. Пример использования:

```
float test = cos( 3.14/2 );
// test = 0
```

sin (x)

Возвращает синус угла x, выраженного в радианах. Пример использования:

```
float test = sin( 3.14/2 );
// test = 1
```

radians (x)

Возвращает количество радиан, равное x градусам. Пример использования:

```
float rad = radians( 180 );
// rad содержит число Пи
```

degrees (x)

Возвращает количество градусов, равное x радианам. Пример использования:

```
float deg = degrees( 3.14/2 );
// deg содержит 90
```

Теперь, зная, как объявляются переменные и пишутся функции, а также имея представление о встроенных функциях, мы уже сможем попытаться написать наш…

Первый эффект и первые шейдеры

Для начала определимся, что мы хотим от наших шейдеров. Наверное, для первого опыта не стоит затевать ничего грандиозного, поэтому задание для себя определим следующим образом: вводить будем только функции для вершинного и пиксельного шейдеров, геометрического шейдера пока не будет. Вершинный шейдер не будет производить никаких изменений с входными данными, они сразу же передаются на следующую стадию конвейера. Пиксельный шейдер будет устанавливать цвет всех пикселов в белый цвет без прозрачности. Исходный код нашего первого шейдера представлен в листинге 5.1. Открываем "Блокнот", набираем исходный текст и сохраняем его в файл под именем first.fx (файл с текстом можно найти на прилагающемся компакт-диске в папке Glava5\FirstFX). Практически весь текст шейдера вопросов уже вызывать не должен, но все же разберем его поподробнее.

Листинг 5.1

```
// Первые шейдеры для Direct3D 10
// Файл first.fx
//--------------------------------------------------------------------------
// Структуры данных
//--------------------------------------------------------------------------
struct VS_INPUT
{
    float4 Pos : POSITION;
};

struct PS_INPUT
{
    float4 Pos : SV_POSITION;
};
//--------------------------------------------------------------------------
// Функция вершинного шейдера
//--------------------------------------------------------------------------
PS_INPUT VS( VS_INPUT Data )
{
    PS_INPUT Out;
    Out.Pos = Data.Pos;
    return Out;
}
//--------------------------------------------------------------------------
// Функция пиксельного шейдера
//--------------------------------------------------------------------------
float4 PS( PS_INPUT Pos ) : SV_Target
{
    return float4( 1.0f, 1.0f, 1.0f, 1.0f );// Белый цвет
}
//--------------------------------------------------------------------------
// Техника отображения
//--------------------------------------------------------------------------
technique10 RenderWhite
{
    pass P0
    {
```

```
        SetVertexShader( CompileShader( vs_4_0, VS() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, PS() ) );
    }
}
```

В первую очередь мы определяем структуры данных, которыми будем пользоваться. У нас это структуры `VS_INPUT` и `PS_INPUT`, которые представляют собой входные данные вершинного и пиксельного шейдеров, соответственно. Конечно, можно было обойтись и без структур, а пользоваться типом `float4`, но дело в том, что такой минимальный набор параметров встречается разве что в учебных программах. Поэтому лучше сразу настроим себя на использование структур, ведь в дальнейших главах параметров у нас будет больше.

Далее мы объявляем функцию вершинного шейдера под именем `VS`. Функция принимает один параметр типа `VS_INPUT`, то есть структуру, которую мы задали несколькими строками выше. Возвращает значения функция также в структуре, на этот раз `PS_INPUT` (для вершинного шейдера это выходные данные, и одновременно входные для пиксельного). В теле функции объявлена переменная `Out`, это структура типа `PS_Input`, в которой функция вернет результат. Члену `Pos` структуры `Out` присваивается значение `Pos` структуры типа `VS_Input` входного параметра. После этого переменная `Out` возвращается оператором `return` как результат работы вершинного шейдера.

За функцией вершинного шейдера следует функция пиксельного шейдера. Она принимает один параметр типа `PS_INPUT`. Тело функции содержит единственную строку с оператором `return`, возвращаемое значение не зависит от параметров функции и всегда представляет собой белый цвет без прозрачности.

После всех функций шейдеров записывается описание техники отображения. В первой строке после ключевого слова `technique10` указывается имя техники, которую мы назвали `RenderWhite`, так как у нас на выходе получаются исключительно белые пикселы. За именем техники отображения следует ее непосредственное описание, заключенное в фигурные скобки. Техника отображения может состоять из нескольких проходов. Например, при необходимости вывода двухмерных спрайтов поверх трехмерной графики, при первом проходе выводится трехмерная геометрия, а во втором проходе рисуются спрайты. В примере, приведенном в листинге 5.1, мы используем только один проход.

Описание техники отображения начинается с ключевого слова `pass`, за которым следует имя прохода, а за ним в фигурных скобках назначаются шейдеры.

Первым назначается вершинный шейдер. На человеческом языке эта строка звучала бы так: скомпилировать функцию VS с использованием модели шейдера vs_4_0 и назначить ее вершинным шейдером. В следующей строке указывается, что геометрический шейдер у нас пока отсутствует, а в следующей за ней назначается пиксельный шейдер. Его назначение происходит так же, как и назначение вершинного шейдера.

Итак, первый файл эффектов записан, первые шейдеры составлены. Самое время нам отправиться к следующей главе, в которой мы начнем, наконец, близкое знакомство с программированием трехмерной графики с помощью Direct3D 10.

Глава 6

Запускаем Direct3D 10

Мы наконец-то подошли к главе о работе с трехмерной графикой. В книгах по программированию графики первая программа с использованием какой-либо графической библиотеки обычно выводит на экран треугольник. Оно и понятно: треугольник в трехмерной графике — фигура основная. Давайте тоже начнем с рисования треугольника, заодно посмотрим в работе наши шейдеры, написанные *в предыдущей главе*.

Рисуем треугольник

Прежде чем на экране мы увидим нужную нам фигуру, нам придется выполнить некоторые подготовительные операции. Итак, предстоит сделать следующее:

- ❒ Описать формат вершины.
- ❒ Создать эффект из файла .FX, который содержит шейдеры для выполнения соответствующих стадий графического конвейера (вершинный, пиксельный и, если необходимо, геометрический шейдеры).
- ❒ Извлечь из созданного эффекта технику отображения.
- ❒ Описать формат входных данных.
- ❒ Создать объект входных данных.
- ❒ Связать объект входных данных с графическим конвейером.
- ❒ Создать буфер вершин.
- ❒ Связать буфер вершин с графическим конвейером.
- ❒ Задать тип и способ построения примитивов.

- В функции визуализации трехмерной сцены осуществить прорисовку по данным из вершинного буфера.
- Добавить соответствующее освобождение ресурсов.

Прокомментируем некоторые моменты, с которыми мы встретились впервые. Создать эффект из файла означает загрузить в память и скомпилировать исходный текст шейдеров, находящихся в файле эффектов. Из этого файла нам еще понадобится извлечь технику отображения, она нам потребуется для создания объекта входных данных.

Разберемся, для чего нужно описывать формат вершины. Чтобы нарисовать треугольник, мы должны сообщить процессору видеокарты информацию о положении трех его вершин. Как передать эту информацию? В Direct3D 10 все данные о вершинах хранятся в буферном ресурсе. Буфер, который содержит координаты и описание прочих свойств вершин, называется *вершинным буфером*. Для создания буферного ресурса мы должны указать размер буфера данных в байтах, а для этого, в свою очередь, нам нужно знать, сколько байт требуется для хранения информации об одной вершине. Если мы это знаем, размер вершинного буфера легко вычисляется умножением количества вершин на количество байт на одну вершину.

Именно для того, чтобы узнать количество байт на одну вершину, и описывается формат вершины. Формат определяет, какие имеются характеристики у вершины (координаты положения в пространстве, цвет, нормаль, текстурные координаты и т. д.), в каком порядке они располагаются в памяти и какой тип данных используют. Обычно формат вершины представляют в виде структуры, а потом определяют ее размер с помощью оператора `sizeof()`.

Хорошо, мы вычислили размер вершинного буфера в байтах и создали буферный ресурс. Теперь мы можем передавать данные видеокарте, но для нее все байты в этом буфере на одно лицо. Неизвестно, где кончаются байты, которые задают координаты вершины, а где начинаются данные о цвете. Процессору видеокарты, помимо собственно вершинного буфера, требуется информация, как эти данные "расшифровать". То есть чтобы он "знал", например, что вот эти первые двенадцать байт — это координаты вершины в формате `float`, за которыми следуют шестнадцать байт, где закодирован цвет, и т. д. Чтобы "ввести в курс дела" процессор видеокарты, мы и сообщаем ему формат входных данных.

Сам буфер вершин представляет собой обычный массив, заполненный данными о координатах, цвете и остальных характеристиках в соответствии с объявленным ранее форматом вершины. Этот массив используется для создания (заполнения) буферного ресурса, который, в свою очередь, связывается с графическим конвейером.

Для правильного построения примитивов по данным из вершинного буфера необходимо указать способ их построения. Способ построения определяет, каким образом выбираются вершины для построения примитивов. Рассмотрим два способа построения треугольников. Первый из них — самый очевидный (рис. 6.1): отбираются три вершины, следующие по порядку, они образуют первый треугольник, следующие три — второй треугольник и т. д. Этот тип построения соответствует значению `D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST` (список треугольников). В отличие от следующего способа, здесь каждая вершина используется один раз.

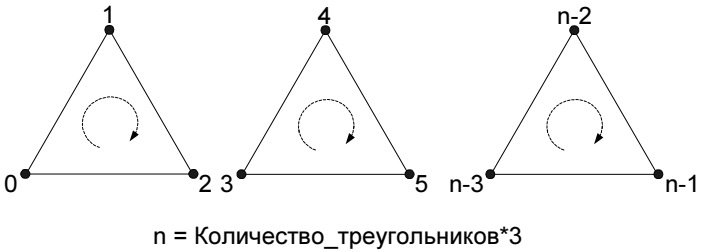

Рис. 6.1. Построение списка треугольников

Второй способ позволяет строить связанные между собой треугольники. При этом для построения первого треугольника отбираются три вершины, для второго берется одна новая вершина, а другие две — от предыдущего треугольника и т. д. (рис. 6.2). Такой тип построения соответствует значению `D3D10_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP` (полоса треугольников).

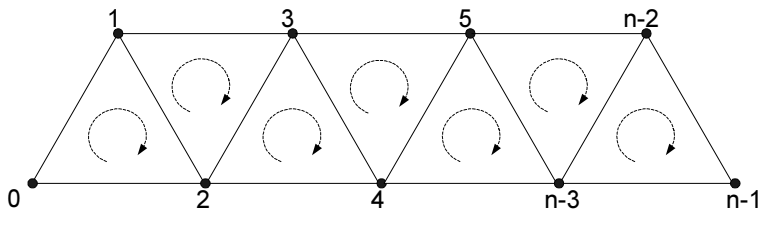

Рис. 6.2. Построение полосы треугольников

Конечно, кроме треугольников есть и другие примитивы, и у них есть свои, похожие способы построения. Мы рассмотрели только два способа построения треугольников для понимания самой идеи.

Перейдем к практике. Создадим новый проект, скопируем в него исходный текст минимального приложения Direct3D 10 (см. листинг 2.2). Сначала давайте определимся с форматом вершины: в нашей программе мы будем задавать пока только координаты положения вершины в пространстве. Опишем структуру, содержащую формат вершины, ее описание будет располагаться перед объявлением глобальных переменных. Тут все довольно просто:

```
struct SimpleVertex
{
  D3DXVECTOR3 Pos;
};
```

На всякий случай расшифруем: в структуре содержится один трехмерный вектор `Pos`. Здесь пока все, переключимся на глобальные переменные. Для рисования треугольника нам потребуется добавить четыре новых переменных:

- указатель на интерфейс работы с эффектами

  ```
  ID3D10Effect*           g_pEffect = NULL;
  ```

- указатель на интерфейс техники отображения

  ```
  ID3D10EffectTechnique*  g_pTechnique = NULL;
  ```

- указатель на интерфейс объекта входных данных

  ```
  ID3D10InputLayout*      g_pVertexLayout = NULL;
  ```

- указатель на интерфейс буферного ресурса

  ```
  ID3D10Buffer*           g_pVertexBuffer = NULL;
  ```

Затем мы открываем функцию `InitDirect3D10()`, от начала и до настройки области отображения она нас полностью устраивает. После настройки области отображения мы начинаем вписывать новые строки. Первое, что мы делаем, это создаем эффект из файла first.fx. Кстати говоря, этот файл должен находиться в той же самой папке, что и исполняемый файл. Сразу скопируйте его в папку с проектом, чтобы потом не забыть. Вот как мы создаем эффект:

```
hr = D3DX10CreateEffectFromFile( L"first.fx", NULL, NULL, "fx_4_0",
          D3D10_SHADER_ENABLE_STRICTNESS, 0, g_pd3dDevice, NULL, NULL,
          &g_pEffect, NULL, NULL );
    if( FAILED( hr ) )
    {
        MessageBox( NULL, L"Не удается обнаружить файл
эффектов (FX). Файл эффектов должен находиться в той же папке,
что и исполняемый файл", L"Ошибка", MB_OK );
        return hr;
    }
```

А именно мы вызываем функцию `D3DX10CreateEffectFromFile()` и проверяем результат ее выполнения. Если функция завершилась с ошибкой — выходим из функции `InitDirect3D10()`. Традиционно рассмотрим прототип функции, чтобы уяснить значение параметров:

```
HRESULT D3DX10CreateEffectFromFile(
  LPCSTR pFileName,
  CONST D3D10_SHADER_MACRO *pDefines,
  ID3D10Include *pInclude,
  LPCSTR pProfile,
  UINT HLSLFlags,
  UINT FXFlags,
  ID3D10Device *pDevice,
  ID3D10EffectPool *pEffectPool,
  ID3DX10ThreadPump *pPump,
  ID3D10Effect **ppEffect,
  ID3D10Blob **ppErrors
  HRESULT *pHResult
);
```

Вот что означают параметры:

- `pFileName` — имя файла эффектов в символах ASCII;
- `pDefines` — массив с завершающим нулем, содержащий описания макросов шейдера, которые используются в файле эффектов (может иметь значение `NULL`);
- `pInclude` — указатель на интерфейс, который обрабатывает подключаемый файл, необходимый для успешной компиляции (может иметь значение `NULL`);
- `pProfile` — профиль шейдера, параметр определяет модель шейдеров, которая будет использоваться при компиляции файла эффектов (в Windows Vista всегда используется четвертая модель: `fx_4_0`);
- `HLSLFlags` — параметры (флаги) компиляции, относящиеся к шейдерам и типам данных;
- `FXFlags` — параметры (флаги) компиляции, относящиеся к эффектам;
- `pDevice` — указатель на устройство Direct3D 10, к которому будет относиться создаваемый ресурс эффекта;
- `pEffectPool` — указатель на пул эффектов, в котором будет содержать данный эффект, или значение `NULL`, если, как у нас, пул эффектов не используется;

- `pPump` — указатель на интерфейс обработки потоков для организации асинхронной работы; для указания на то, что функция не должна возвращать управление, пока она не завершилась, ставим значение `NULL`;
- `ppEffect` — адрес указателя на создаваемый эффект;
- `ppErrors` — адрес указателя на область памяти, в которой содержится описание ошибок, возникших при компиляции, если, конечно, они возникли;
- `pHResult` — указатель на переменную, в которой возвращается код завершения функции.

Заданные нами параметры практически не нуждаются в комментариях, так как многие из них имеют либо нулевое значение, либо значение `NULL`. Следует пояснить, пожалуй, только флаг `D3D10_SHADER_ENABLE_STRICTNESS`, он означает, что при компиляции шейдера будет запрещено использование устаревшего синтаксиса из предыдущих версий HLSL.

Остановимся поподробнее на просмотре сообщений об ошибках, возникших при компиляции шейдеров. В нашем примере мы такую возможность не используем, но в дальнейшем она вам может понадобиться. Сами сообщения можно прочитать с помощью отладчика Visual Studio, но для этого в программу необходимо кое-что добавить. Измененный фрагмент текста программы выглядит следующим образом:

```
ID3D10Blob* l_pBlob_Errors = NULL;
hr = D3DX10CreateEffectFromFile( L"first.fx", NULL, NULL, "fx_4_0",
          D3D10_SHADER_ENABLE_STRICTNESS, 0, g_pd3dDevice, NULL, NULL,
          &g_pEffect, &l_pBlob_Errors, NULL );
// Указатель
LPVOID l_pError = NULL;
// Если были ошибки — получаем указатель на текст сообщений
if( l_pBlob_Errors )
{
    l_pError = l_pBlob_Errors->GetBufferPointer();
    // Теперь l_pError указывает на текстовое сообщение
}
```

Перед вызовом функции создания эффекта объявляем указатель на интерфейс `ID3D10Blob`. Этот интерфейс используется для вывода данных произвольной длины, в нашем случае — для вывода сообщений об ошибках. После объявления указателя `l_pBlob_Errors` мы вызываем функцию создания эффекта, в которой предпоследним параметром идет его адрес. После вызова функции мы объявляем указатель `l_pError` типа `LPVOID`. Если в процессе

компиляции шейдеров возникли ошибки, то указателю `l_pError` будет присвоен адрес начала текста сообщений об ошибках. Этот адрес мы получаем, вызвав метод `ID3D10Blob::GetBufferPointer`. Сам текст можно прочитать в окне отслеживаемых переменных (watches) Visual Studio, для этого в список отслеживаемых значений нужно добавить выражение `(char*)l_pError`, поскольку сам указатель имеет тип `LPVOID`.

Продолжим писать нашу программу. Извлечем технику отображения из созданного эффекта. Для этого мы воспользуемся методом `ID3D10Effect::GetTechniqueByName`, в качестве параметра указываем имя единственной техники отображения, которое у нас имеется: "RenderWhite". Вот как вызов метода выглядит в программе:

```
g_pTechnique = g_pEffect->GetTechniqueByName( "RenderWhite" );
```

Теперь мы описываем формат входных данных. Описание содержится в массиве типа `D3D10_INPUT_ELEMENT_DESC`, каждый элемент которого представляет ту или иную характеристику вершины. Как мы договорились, у нас будет только одна характеристика — координаты вершины в пространстве, значит, и элемент в массиве будет только один. Вот так выглядит объявление массива:

```
D3D10_INPUT_ELEMENT_DESC layout[] =
{
 { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
D3D10_INPUT_PER_VERTEX_DATA, 0 },
};
```

Здесь пока ничего не понятно. Чтобы разобраться, рассмотрим, что собой представляет каждое поле в структуре `D3D10_INPUT_ELEMENT_DESC`, которая описывается таким образом:

```
typedef struct D3D10_INPUT_ELEMENT_DESC {
    LPCSTR SemanticName;
    UINT SemanticIndex;
    DXGI_FORMAT Format;
    UINT InputSlot;
    UINT AlignedByteOffset;
    D3D10_INPUT_CLASSIFICATION InputSlotClass;
    UINT InstanceDataStepRate;
} D3D10_INPUT_ELEMENT_DESC;
```

Начнем по порядку, с самого начала:

- `SemanticName` — семантика HLSL, которая связывает данный элемент с входным параметром шейдера с такой же семантикой;

- `SemanticIndex` — дополнительный семантический индекс элемента, который используется только в случаях, когда необходимо передавать несколько элементов с одинаковой семантикой, этот индекс меняет семантическое имя, добавляя к нему указанный индекс;
- `Format` — тип данных (формат) элемента;
- `InputSlot` — номер входного канала конвейера, допустимые значения 0 — 15;
- `AlignedByteOffset` — необязательный параметр, представляет собой смещение описываемого элемента в байтах от начала данных вершины. Информация о вершине для процессора видеокарты представляет собой обычный фрагмент памяти, а смещение показывает, с какого байта начинать считывание элемента массива. Чтобы не высчитывать смещение самому, а просто указать, что элемент следует сразу за предыдущим, можно использовать константу `D3D10_APPEND_ALIGNED_ELEMENT`;
- `InputSlotClass` — определяет тип данных, передаваемых по входному каналу, мы используем значение `D3D10_INPUT_PER_VERTEX_DATA`, которое указывает, что с помощью данного элемента передаются данные, характеризующие одну вершину;
- `InstanceDataStepRate` — используется при создании экземпляров (instancing), в нашем случае устанавливаем значение поля в 0.

После заполнения массива мы вычислим количество элементов, в нем находящихся:

```
UINT numElements = sizeof(layout)/sizeof(layout[0]);
```

Конечно, сейчас у нас один элемент, вроде бы, чего тут считать? Присвоить переменной значение 1, и дело с концом! Тем не менее, если в дальнейшем нам понадобится добавить в массив `layout[]` новые элементы, получившееся количество элементов посчитается автоматически, а это убережет нас от дополнительных исправлений текста программы и, следовательно, от дополнительных ошибок.

Следующее, что мы сделаем, это создадим объект входных данных. Для создания этого объекта мы будем использовать метод `ID3D10Device::CreateInputLayout`, рассмотрим его прототип, попутно отмечая, данных для каких параметров нам не хватает:

```
HRESULT CreateInputLayout(
  const D3D10_INPUT_ELEMENT_DESC *pInputElementDescs,
  UINT NumElements,
  const void *pShaderBytecodeWithInputSignature,
```

```
  SIZE_T BytecodeLength,
  ID3D10InputLayout **ppInputLayout
);
```

Подробнее опишем параметры:

- `pInputElementDescs` — массив, содержащий описания типов данных, передаваемых на первую стадию графического конвейера, то есть тот массив, в котором мы закодировали формат входных данных, массив `layout[]`;
- `NumElements` — количество описаний типов данных, то есть — количество элементов в массиве `layout[]`;
- `pShaderBytecodeWithInputSignature` — указатель на сигнатуру входных данных шейдера;
- `BytecodeLength` — размер сигнатуры в байтах;
- `ppInputLayout` — адрес указателя, в котором будет возвращен созданный объект входных данных.

Выясняется, что для вызова метода у нас уже есть практически все необходимое, за исключением указателя на какую-то там сигнатуру. Что такое сигнатура, да еще и не просто сигнатура, а сигнатура входных данных шейдера? Для наших целей достаточно понимать, что сигнатура — это некий идентификатор шейдера, однозначно связанный с типами и количеством его входных параметров. То есть если имеется два совершенно разных по внутреннему содержанию шейдера, а на входе они принимают одни и те же параметры, которые к тому же имеют одинаковый тип, то эти шейдеры имеют одинаковую сигнатуру входных данных. Возникает вопрос: откуда ее взять? К счастью для нас с вами, ответ на него совсем не сложен, поскольку шейдеры у нас уже откомпилированы. С помощью извлеченной из шейдера техники отображения мы получим указатель на интерфейс первого прохода (с индексом 0) и для него вызовем метод получения описания прохода, чтобы заполнить структуру типа `D3D10_PASS_DESC`. А в этой структуре уже есть поля, содержащие нужные нам значения. Описанный способ займет всего две строчки исходного текста:

```
D3D10_PASS_DESC PassDesc;
g_pTechnique->GetPassByIndex( 0 )->GetDesc( &PassDesc );
```

Подробно рассматривать структуру `D3D10_PASS_DESC` мы не будем, а воспользуемся значениями двух ее полей, `pIAInputSignature` и `IAInputSignatureSize`, которые содержат указатель на сигнатуру входных данных и размер сигнатуры в байтах, соответственно. Теперь для создания объекта входных данных у нас нет никаких преград:

```
hr = g_pd3dDevice->CreateInputLayout( layout, numElements,
          PassDesc.pIAInputSignature, PassDesc.IAInputSignatureSize,
```

```
        &g_pVertexLayout );
    if( FAILED( hr ) )
        return hr;
```

Объект входных данных создан, и теперь нужно связать его с графическим конвейером. Для этого мы вызовем специальный метод `ID3D10Device::IASetInputLayout`, который требует наличия единственного параметра — указателя на объект входных данных:

```
g_pd3dDevice->IASetInputLayout( g_pVertexLayout );
```

Затем нам нужно создать буфер вершин. Буфер вершин создается вызовом метода `ID3D10Device::CreateBuffer`. Рассмотрим его прототип:

```
HRESULT CreateBuffer(
  const D3D10_BUFFER_DESC *pDesc,
  const D3D10_SUBRESOURCE_DATA *pInitialData,
  ID3D10Buffer **ppBuffer
);
```

- `pDesc` — указатель на структуру типа `D3D10_BUFFER_DESC`, содержащую описание буфера;
- `pInitialData` — указатель на данные, которыми будет инициализирован (заполнен) буфер;
- `ppBuffer` — адрес указателя, который будет указывать на созданный буфер.

Начнем с того, что создадим массив типа `SimpleVertex` с численными значениями координат вершин треугольника:

```
SimpleVertex vertices[] =
{
  D3DXVECTOR3( 0.0f, 0.5f, 0.5f ),
  D3DXVECTOR3( 0.5f,-0.5f, 0.5f ),
  D3DXVECTOR3(-0.5f,-0.5f, 0.5f ),
};
```

Теперь подготовим почву для создания самого буфера, для этого нам необходимо заполнить структуры `D3D10_BUFFER_DESC` и `D3D10_SUBRESOURCE_DATA`. Как нам уже известно, первая представляет собой описание свойств создаваемого буфера, а вторая — описание данных, которые будут в него скопированы в процессе создания.

Замечание

Вам уже наверняка надоели все эти бесконечные структуры, поля, параметры… Здесь ничего не поделаешь, нужно все это усвоить. За счет большого количества параметров достигается большая гибкость использования графиче-

ской библиотеки. То есть вещи это все нужные, крепитесь, оно того стоит. К тому же, осталось не так уж и много.

Рассмотрим прототип структуры `D3D10_BUFFER_DESC`:

```
typedef struct D3D10_BUFFER_DESC {
    UINT ByteWidth;
    D3D10_USAGE Usage;
    UINT BindFlags;
    UINT CPUAccessFlags;
    UINT MiscFlags;
} D3D10_BUFFER_DESC;
```

Разберем значение полей:

- `ByteWidth` — размер буфера в байтах (именно ради этого поля мы в самом начале объявили структуру `SimpleVertex`, теперь для вычисления размера буфера нам нужно взять ее размер в байтах и умножить на количество вершин, а в треугольнике, как вы понимаете, их ровно три);
- `Usage` — режим использования буфера, зависит от того, как требуется производить чтение из буфера и запись в него, ключевым фактором является частота обновления данных в буфере (обычно используется флаг `D3D10_USAGE_DEFAULT`);
- `BindFlags` — флаги, определяющие, в каком качестве буфер связывается с графическим конвейером, как обычно, несколько флагов можно объединить с помощью операции OR (в нашем случае флаг только один: `D3D10_BIND_VERTEX_BUFFER`, что означает связывание в качестве вершинного буфера);
- `CPUAccessFlags` — флаги, указывающие, какие виды доступа разрешаются для центрального процессора, поле может содержать несколько объединенных с помощью операции OR флагов (если доступ для центрального процессора не требуется, устанавливается значение 0);
- `MiscFlags` — определяет прочие, реже используемые свойства буфера, поле также может содержать несколько объединенных с помощью операции OR флагов, если не требуется использование ни одного из флагов, устанавливается значение 0.

Познакомившись с этой структурой, мы можем вполне осмысленно ее заполнить:

```
D3D10_BUFFER_DESC bd;
bd.Usage = D3D10_USAGE_DEFAULT;
bd.ByteWidth = sizeof( SimpleVertex ) * 3;
```

```
bd.BindFlags = D3D10_BIND_VERTEX_BUFFER;
bd.CPUAccessFlags = 0;
bd.MiscFlags = 0;
```

Прототип структуры `D3D10_SUBRESOURCE_DATA` мы рассматривать не будем, так как у нее мы установим значение лишь одного-единственного поля. Значения двух других полей используются только в том случае, если производится работа с текстурой, а в нашем случае эти значения игнорируются. Мы присваиваем значение полю `pSysMem`, представляющее собой указатель на данные, которыми будет заполнен создаваемый буфер вершин. Этими данными, конечно, будет содержимое массива `vertices[]`:

```
D3D10_SUBRESOURCE_DATA InitData;
InitData.pSysMem = vertices;
```

Создаем буфер вершин:

```
hr = g_pd3dDevice->CreateBuffer( &bd, &InitData, &g_pVertexBuffer );
if( FAILED( hr ) )
        return hr;
```

Созданный буфер вершин нужно связать с графическим конвейером. Для выполнения этой задачи существует метод `ID3D10Device::IASetVertexBuffers`, прототип которого приведен ниже:

```
void IASetVertexBuffers(
  UINT StartSlot,
  UINT NumBuffers,
  ID3D10Buffer *const *ppVertexBuffers,
  const UINT *pStrides,
  const UINT *pOffsets
);
```

При вызове методу передаются следующие параметры:

- `StartSlot` — индекс первого входного канала для связывания. Первый вершинный буфер в массиве (см. ниже) явно связывается с каналом, установленном в этом параметре, каждый последующий буфер связывается со следующим по возрастанию индексов каналом, максимально можно задействовать 16 каналов;
- `NumBuffers` — количество вершинных буферов в массиве, которое не может превышать максимального количества входных каналов;
- `ppVertexBuffers` — указатель на массив из буферов вершин, буферы должны быть созданы с флагом `D3D10_BIND_VERTEX_BUFFER`;
- `pStrides` — указатель на массив, содержащий для каждого буфера вершин величину шага для перехода к следующему элементу. Шаг представляет собой размер элемента соответствующего буфера вершин в байтах;

- `pOffsets` — указатель на массив, содержащий значения смещений для каждого буфера вершин. Смещение — это количество байт между первым элементом в буфере и первым элементом из этого буфера, который будет использован.

У нас имеется только один буфер вершин, подберем параметры исходя из этого. Определим шаг для перехода к следующему элементу:

```
UINT stride = sizeof( SimpleVertex );
```

Зададим смещение для нашего буфера вершин. У нас используются все элементы буфера, начиная с самого начала, поэтому смещение равно нулю:

```
UINT offset = 0;
```

Связываем буфер вершин с графическим конвейером, устанавливаем нулевой входной канал, а количество буферов вершин задаем равным единице:

```
g_pd3dDevice->IASetVertexBuffers( 0, 1, &g_pVertexBuffer,
                                  &stride, &offset );
```

Все практически готово, осталось определить, какие примитивы и каким образом будут строиться по координатам вершин. Строить мы собрались треугольник, а способ построения зададим как "список треугольников". Для этого вызовем метод `ID3D10Device::IASetPrimitiveTopology`, который требует указания одного параметра, задающего тип и способ построения примитивов. Нам нужно задать значение `D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST`:

```
g_pd3dDevice->IASetPrimitiveTopology(
                    D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST );
```

Ну, вот и все. Работу с функцией `InitDirect3D10()` мы наконец-то закончили. Давайте еще раз пробежим ее взглядом с самого начала и до конца:

```
HRESULT InitDirect3D10()
{
       HRESULT hr = S_OK;

       // Размеры клиентской области окна
       RECT rc;
       GetClientRect( g_hWnd, &rc );
       UINT width = rc.right - rc.left;
       UINT height = rc.bottom - rc.top;

       // Список возможных типов устройства
       D3D10_DRIVER_TYPE driverTypes[] =
       {
              D3D10_DRIVER_TYPE_HARDWARE,
```

```
            D3D10_DRIVER_TYPE_REFERENCE,
        };
        UINT numDriverTypes =
sizeof(driverTypes) / sizeof(driverTypes[0]);

        // Заполняем структуру
        DXGI_SWAP_CHAIN_DESC sd;
        ZeroMemory( &sd, sizeof(sd) );
        sd.BufferCount = 1;
        sd.BufferDesc.Width = width;
        sd.BufferDesc.Height = height;
        sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
        sd.BufferDesc.RefreshRate.Numerator = 60;
        sd.BufferDesc.RefreshRate.Denominator = 1;
        sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
        sd.OutputWindow = g_hWnd;
        sd.SampleDesc.Count = 1;
        sd.SampleDesc.Quality = 0;
        sd.Windowed = TRUE;

        // Пытаемся создать устройство, проходя по списку
        // Как только получилось - выходим из цикла
        for( UINT driverTypeIndex = 0;
                driverTypeIndex < numDriverTypes; driverTypeIndex++ )
        {
          g_driverType = driverTypes[driverTypeIndex];
          hr = D3D10CreateDeviceAndSwapChain( NULL, g_driverType, NULL, 0,
                    D3D10_SDK_VERSION, &sd, &g_pSwapChain, &g_pd3dDevice );
          if( SUCCEEDED( hr ) )
                 break;
        }
        if( FAILED(hr) )
               return hr;

        // Представление данных для
        // буфера визуализации
        ID3D10Texture2D *pBackBuffer;
        hr = g_pSwapChain->GetBuffer( 0, __uuidof( ID3D10Texture2D ),
                                      (LPVOID*)&pBackBuffer );
```

```
        if( FAILED(hr) )
                return hr;
        // Создадим представление данных
        hr = g_pd3dDevice->CreateRenderTargetView( pBackBuffer, NULL,
                                                   &g_pRenderTargetView );
        pBackBuffer->Release();
        if( FAILED(hr) )
                return hr;
        // Свяжем буфер визуализации с графическим конвейером
        g_pd3dDevice->OMSetRenderTargets( 1, &g_pRenderTargetView, NULL );

        // Настроим область отображения
        D3D10_VIEWPORT vp;
        vp.Width = width;
        vp.Height = height;
        vp.MinDepth = 0.0f;
        vp.MaxDepth = 1.0f;
        vp.TopLeftX = 0;
        vp.TopLeftY = 0;
        g_pd3dDevice->RSSetViewports( 1, &vp );

        // Создадим эффект,
        // используем наши первые шейдеры
        // в файле эффектов first.fx
        hr = D3DX10CreateEffectFromFile( L"first.fx", NULL, NULL,
              "fx_4_0", D3D10_SHADER_ENABLE_STRICTNESS, 0, g_pd3dDevice,
              NULL, NULL, &g_pEffect, NULL, NULL );
        if( FAILED( hr ) )
        {
                MessageBox( NULL, L"Не удается обнаружить файл
эффектов (FX). Файл эффектов должен находиться в той же папке, что
и исполняемый файл", L"Ошибка", MB_OK );
                return hr;
        }

        // Извлекаем технику отображения
        g_pTechnique = g_pEffect->GetTechniqueByName( "RenderWhite" );

        // Описываем формат входных данных
```

```
D3D10_INPUT_ELEMENT_DESC layout[] =
{
       { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
         D3D10_INPUT_PER_VERTEX_DATA, 0 },
};
UINT numElements = sizeof(layout)/sizeof(layout[0]);

// Создаем объект входных данных
D3D10_PASS_DESC PassDesc;
g_pTechnique->GetPassByIndex( 0 )->GetDesc( &PassDesc );
hr = g_pd3dDevice->CreateInputLayout( layout, numElements,
    PassDesc.pIAInputSignature, PassDesc.IAInputSignatureSize,
    &g_pVertexLayout );
if( FAILED( hr ) )
       return hr;

// Связываем объект входных данных с графическим конвейером
g_pd3dDevice->IASetInputLayout( g_pVertexLayout );

// Создаем буфер вершин
SimpleVertex vertices[] =
{
       D3DXVECTOR3( 0.0f, 0.5f, 0.5f ),
       D3DXVECTOR3( 0.5f, -0.5f, 0.5f ),
       D3DXVECTOR3( -0.5f, -0.5f, 0.5f ),
};
D3D10_BUFFER_DESC bd;
bd.Usage = D3D10_USAGE_DEFAULT;
bd.ByteWidth = sizeof( SimpleVertex ) * 3;
bd.BindFlags = D3D10_BIND_VERTEX_BUFFER;
bd.CPUAccessFlags = 0;
bd.MiscFlags = 0;
D3D10_SUBRESOURCE_DATA InitData;
InitData.pSysMem = vertices;
hr = g_pd3dDevice->CreateBuffer(&bd, &InitData, &g_pVertexBuffer);
if( FAILED( hr ) )
       return hr;

// Связываем буфер вершин с графическим конвейером
```

```
        UINT stride = sizeof( SimpleVertex );
        UINT offset = 0;
        g_pd3dDevice->IASetVertexBuffers( 0, 1, &g_pVertexBuffer,
                                          &stride, &offset );

        // Задаем тип и способ построения примитивов
        g_pd3dDevice->IASetPrimitiveTopology(
                        D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST );

        return S_OK;
}
```

После выполнения этой функции буфер вершин у нас создан, процессор видеокарты будет знать, как и какие данные в нем находятся, осталось только отдать команду на рисование. По сравнению с работой, которую мы только что проделали для инициализации Direct3D 10, это все равно, что легкая прогулка. Открываем функцию `RenderScene()`. Все рисование происходит между очисткой вторичного буфера и переключением буферов в конце функции. Рассмотрим все изменения сразу, ввиду малого их объема:

```
D3D10_TECHNIQUE_DESC techDesc;
g_pTechnique->GetDesc( &techDesc );
for( UINT p = 0; p < techDesc.Passes; ++p )
{
        g_pTechnique->GetPassByIndex( p )->Apply(0);
        g_pd3dDevice->Draw( 3, 0 );
}
```

Что же это все означает? Сначала мы объявляем переменную типа `D3D10_TECHNIQUE_DESC`, с ее помощью мы получим количество проходов в технике отображения. В следующей строке мы вызываем метод `ID3D10EffectTechnique::GetDesc`, он заполняет эту структуру параметрами, из которых, как мы условились, нас интересует только один. Далее мы организуем цикл `for`, который выполнится столько раз, сколько проходов содержится в технике отображения. Тело цикла состоит из двух строк.

- В первой мы получаем указатель на интерфейс соответствующего прохода и вызываем метод `ID3D10EffectPass::Apply`. Он переводит устройство в режим отображения, определенный в описании техники в файле эффекта (значение параметра метода игнорируется).
- Во второй строке вызывается метод `ID3D10Device::Draw`, с помощью которого и происходит рисование примитивов, в нашем случае — одного треугольника. Этот метод предназначен для рисования без использования

индексного буфера (как раз наш случай). Первым параметром идет количество вершин, которые нужно нарисовать, вторым параметром указывается индекс вершины, с которой начинать рисование. Мы, как нетрудно догадаться, указываем количество вершин треугольника и нулевой индекс — начинать рисование с первой вершины в буфере.

Вот и все, что нужно добавить для появления треугольника на экране. Приведем функцию `RenderScene()` полностью:

```
void RenderScene()
{
       // Очищаем вторичный буфер
       //(компоненты красного, зеленого, синего, прозрачность)
       float ClearColor[4] = { 0.1f, 0.5f, 0.1f, 1.0f };
g_pd3dDevice->ClearRenderTargetView( g_pRenderTargetView,
ClearColor );

       // Рисуем наш треугольник
       D3D10_TECHNIQUE_DESC techDesc;
       g_pTechnique->GetDesc( &techDesc );
       for( UINT p = 0; p < techDesc.Passes; ++p )
       {
              g_pTechnique->GetPassByIndex( p )->Apply(0);
              g_pd3dDevice->Draw( 3, 0 );
       }
       // Переключаем буферы
       g_pSwapChain->Present( 0, 0 );
}
```

С графической частью мы закончили. Теперь самое время позаботиться об очистке памяти при выходе из программы. Переходим в функцию `Cleanup()` и после первой строки добавляем освобождение ресурсов для вновь введенных интерфейсов:

```
if ( g_pVertexBuffer ) g_pVertexBuffer->Release();
if ( g_pVertexLayout ) g_pVertexLayout->Release();
if ( g_pEffect ) g_pEffect->Release();
```

Полностью функция должна получиться такой:

```
void Cleanup()
{
       if( g_pd3dDevice ) g_pd3dDevice->ClearState();
```

```
        if ( g_pVertexBuffer ) g_pVertexBuffer->Release();
        if ( g_pVertexLayout ) g_pVertexLayout->Release();
        if ( g_pEffect ) g_pEffect->Release();

        if( g_pRenderTargetView ) g_pRenderTargetView->Release();
        if( g_pSwapChain ) g_pSwapChain->Release();
        if( g_pd3dDevice ) g_pd3dDevice->Release();
}
```

Итак, мы закончили нашу первую программу, выводящую графику с использованием буфера вершин. Откомпилируйте и запустите проект. Результат работы приложения можно увидеть на рис. 6.3.

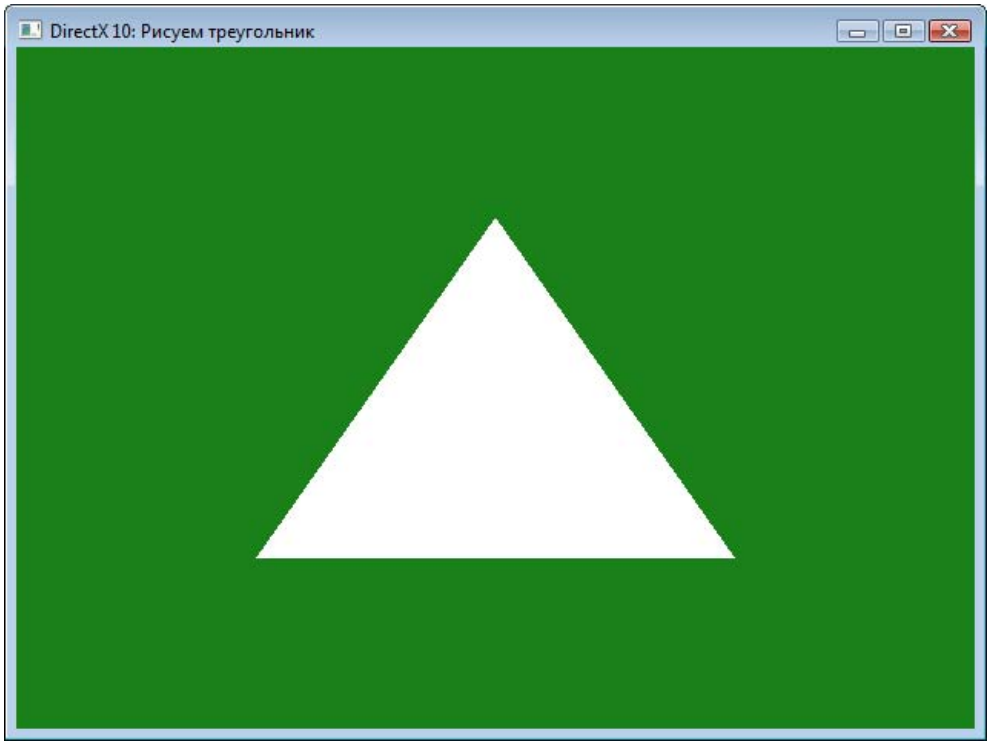

Рис. 6.3. Вывод на экран треугольника

Полный исходный текст приложения представлен в листинге 6.1, соответствующий проект можно найти на компакт-диске в директории Glava6\Triangle.

Листинг 6.1

```
//--------------------------------------------------------------------------
//  Рисуем треугольник
//--------------------------------------------------------------------------

#include <windows.h>
#include <d3d10.h>
#include <d3dx10.h>

// Ширина и высота окна
#define WINDOW_WIDTH  640
#define WINDOW_HEIGHT 480

//--------------------------------------------------------------------------
// Структуры
//--------------------------------------------------------------------------
struct SimpleVertex
{
       D3DXVECTOR3 Pos;
};

//--------------------------------------------------------------------------
// Глобальные переменные
//--------------------------------------------------------------------------
HWND        g_hWnd = NULL;
D3D10_DRIVER_TYPE       g_driverType = D3D10_DRIVER_TYPE_NULL;
ID3D10Device*           g_pd3dDevice = NULL;
IDXGISwapChain*         g_pSwapChain = NULL;
ID3D10RenderTargetView* g_pRenderTargetView = NULL;

ID3D10Effect*           g_pEffect = NULL;
ID3D10EffectTechnique*  g_pTechnique = NULL;
ID3D10InputLayout*      g_pVertexLayout = NULL;
ID3D10Buffer*           g_pVertexBuffer = NULL;

//--------------------------------------------------------------------------
// Прототипы функций
//--------------------------------------------------------------------------
```

```
HRESULT             InitWindow( HINSTANCE hInstance, int nCmdShow );
HRESULT             InitDirect3D10();
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);
void                Cleanup();
void                RenderScene();

//--------------------------------------------------------------------------
// С этой функции начинается выполнение программы
//--------------------------------------------------------------------------
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow )
{
       // Создаем окно приложения
       if( FAILED( InitWindow( hInstance, nCmdShow ) ) )
                     return 0;
       //
       if( FAILED( InitDirect3D10() ) )
       {
              Cleanup();
              return 0;
       }

       // Цикл обработки сообщений
       MSG msg = {0};
       while( WM_QUIT != msg.message  )
       {
                     if( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) )
              {
                     TranslateMessage( &msg );
                     DispatchMessage( &msg );
              }
              else
              {
                     RenderScene();
              }

       }

       Cleanup();
```

```
    return (int) msg.wParam;
}

//--------------------------------------------------------------------------
// Регистрация класса и создание окна
//--------------------------------------------------------------------------
HRESULT InitWindow( HINSTANCE hInstance, int nCmdShow )
{
    // Регистрируем класс окна
    WNDCLASSEX wc;
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style          = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc    = WndProc;
    wc.cbClsExtra     = 0;
    wc.cbWndExtra     = 0;
    wc.hInstance      = hInstance;
    wc.hIcon          = NULL;
    wc.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszMenuName   = NULL;
    wc.lpszClassName  = L"SimpleWindowClass";
    wc.hIconSm        = LoadIcon(NULL, IDI_APPLICATION);
    if( !RegisterClassEx(&wc) )
        return E_FAIL;

    // Создаем окно
    g_hWnd = CreateWindow(
        L"SimpleWindowClass",
        L"DirectX 10: Рисуем треугольник",
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        WINDOW_WIDTH,
        WINDOW_HEIGHT,
        NULL,
        NULL,
        hInstance,
        NULL);
```

```
        // Если не удалось создать окно - выходим из функции
        if( !g_hWnd )
                return E_FAIL;
        // Отображаем окно на экране
        ShowWindow( g_hWnd, nCmdShow );
        UpdateWindow(g_hWnd);

        return S_OK;
}

//--------------------------------------------------------------------------
// Инициализация Direct3D
//--------------------------------------------------------------------------
HRESULT InitDirect3D10()
{
        HRESULT hr = S_OK;

        // Размеры клиентской области окна
        RECT rc;
        GetClientRect( g_hWnd, &rc );
        UINT width = rc.right - rc.left;
        UINT height = rc.bottom - rc.top;

        // Список возможных типов устройства
        D3D10_DRIVER_TYPE driverTypes[] =
        {
                D3D10_DRIVER_TYPE_HARDWARE,
                D3D10_DRIVER_TYPE_REFERENCE,
        };
        UINT numDriverTypes = sizeof(driverTypes) /
                                sizeof(driverTypes[0]);

        // Заполняем структуру
        DXGI_SWAP_CHAIN_DESC sd;
        ZeroMemory( &sd, sizeof(sd) );
        sd.BufferCount = 1;
        sd.BufferDesc.Width = width;
        sd.BufferDesc.Height = height;
        sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
```

```
    sd.BufferDesc.RefreshRate.Numerator = 60;
    sd.BufferDesc.RefreshRate.Denominator = 1;
    sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
    sd.OutputWindow = g_hWnd;
    sd.SampleDesc.Count = 1;
    sd.SampleDesc.Quality = 0;
    sd.Windowed = TRUE;

    // Пытаемся создать устройство, проходя по списку
    // Как только получилось - выходим из цикла
    for( UINT driverTypeIndex = 0; driverTypeIndex < numDriverTypes;
                                                    driverTypeIndex++ )
    {
      g_driverType = driverTypes[driverTypeIndex];
      hr = D3D10CreateDeviceAndSwapChain( NULL, g_driverType, NULL, 0,
               D3D10_SDK_VERSION, &sd, &g_pSwapChain, &g_pd3dDevice );
      if( SUCCEEDED( hr ) )
            break;
    }
    if( FAILED(hr) )
          return hr;

    // Представление данных для
    // буфера визуализации
    ID3D10Texture2D *pBackBuffer;
    hr = g_pSwapChain->GetBuffer( 0, __uuidof( ID3D10Texture2D ),
                                  (LPVOID*)&pBackBuffer );
    if( FAILED(hr) )
          return hr;
// Создадим представление данных
    hr = g_pd3dDevice->CreateRenderTargetView( pBackBuffer, NULL,
                                               &g_pRenderTargetView );
    pBackBuffer->Release();
    if( FAILED(hr) )
          return hr;
// Свяжем буфер визуализации с графическим конвейером
    g_pd3dDevice->OMSetRenderTargets( 1, &g_pRenderTargetView, NULL );

    // Настроим область отображения
```

```
        D3D10_VIEWPORT vp;
        vp.Width = width;
        vp.Height = height;
        vp.MinDepth = 0.0f;
        vp.MaxDepth = 1.0f;
        vp.TopLeftX = 0;
        vp.TopLeftY = 0;
        g_pd3dDevice->RSSetViewports( 1, &vp );

        // Создадим эффект,
        // используем наши первые шейдеры
        // в файле эффектов first.fx
hr = D3DX10CreateEffectFromFile( L"first.fx", NULL, NULL, "fx_4_0",
D3D10_SHADER_ENABLE_STRICTNESS, 0, g_pd3dDevice, NULL, NULL,
&g_pEffect, NULL, NULL );
        if( FAILED( hr ) )
        {
                MessageBox( NULL, L"Не удается обнаружить файл эффектов
(FX). Файл эффектов должен находиться в той же папке, что и исполняемый
файл", L"Ошибка", MB_OK );
                return hr;
        }

        // Извлекаем технику отображения
        g_pTechnique = g_pEffect->GetTechniqueByName( "RenderWhite" );

        // Описываем формат входных данных
        D3D10_INPUT_ELEMENT_DESC layout[] =
        {
                { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
                  D3D10_INPUT_PER_VERTEX_DATA, 0 },
        };
        UINT numElements = sizeof(layout)/sizeof(layout[0]);

        // Создаем объект входных данных
        D3D10_PASS_DESC PassDesc;
        g_pTechnique->GetPassByIndex( 0 )->GetDesc( &PassDesc );
        hr = g_pd3dDevice->CreateInputLayout( layout, numElements,
             PassDesc.pIAInputSignature, PassDesc.IAInputSignatureSize,
```

```
        &g_pVertexLayout );
    if( FAILED( hr ) )
        return hr;

    // Связываем объект входных данных с графическим конвейером
    g_pd3dDevice->IASetInputLayout( g_pVertexLayout );

    // Создаем буфер вершин
    SimpleVertex vertices[] =
    {
        D3DXVECTOR3( 0.0f, 0.5f, 0.5f ),
        D3DXVECTOR3( 0.5f,-0.5f, 0.5f ),
        D3DXVECTOR3(-0.5f,-0.5f, 0.5f ),
    };
    D3D10_BUFFER_DESC bd;
    bd.Usage = D3D10_USAGE_DEFAULT;
    bd.ByteWidth = sizeof( SimpleVertex ) * 3;
    bd.BindFlags = D3D10_BIND_VERTEX_BUFFER;
    bd.CPUAccessFlags = 0;
    bd.MiscFlags = 0;
    D3D10_SUBRESOURCE_DATA InitData;
    InitData.pSysMem = vertices;
    hr = g_pd3dDevice->CreateBuffer( &bd, &InitData,
                                     &g_pVertexBuffer );
    if( FAILED( hr ) )
        return hr;

    // Связываем буфер вершин с графическим конвейером
    UINT stride = sizeof( SimpleVertex );
    UINT offset = 0;
    g_pd3dDevice->IASetVertexBuffers( 0, 1, &g_pVertexBuffer, &stride,
                                      &offset );

    // Задаем тип и способ построения примитивов
    g_pd3dDevice->IASetPrimitiveTopology(
                        D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST );

    return S_OK;
}
```

```
//--------------------------------------------------------------------------
// Прорисовка трехмерной сцены
//--------------------------------------------------------------------------
void RenderScene()
{
	// Очищаем вторичный буфер
	//(компоненты красного, зеленого, синего, прозрачность)
	float ClearColor[4] = { 0.1f, 0.5f, 0.1f, 1.0f };
	g_pd3dDevice->ClearRenderTargetView( g_pRenderTargetView,
                                                    ClearColor );

	// Рисуем наш треугольник
	D3D10_TECHNIQUE_DESC techDesc;
	g_pTechnique->GetDesc( &techDesc );
	for( UINT p = 0; p < techDesc.Passes; ++p )
	{
		g_pTechnique->GetPassByIndex( p )->Apply(0);
		g_pd3dDevice->Draw( 3, 0 );
	}
	// Переключаем буферы
	g_pSwapChain->Present( 0, 0 );
}
//--------------------------------------------------------------------------
// Обработка сообщений
//--------------------------------------------------------------------------
LRESULT CALLBACK WndProc( HWND hWnd, UINT message, WPARAM wParam,
                          LPARAM lParam )
{
	switch (message)
	{

	case WM_DESTROY:
		PostQuitMessage(0);
		break;

	default:
		return DefWindowProc(hWnd, message, wParam, lParam);
	}

	return 0;
```

```
}

//--------------------------------------------------------------------------
// Очищаем память
//--------------------------------------------------------------------------
void Cleanup()
{
       if( g_pd3dDevice ) g_pd3dDevice->ClearState();

       if ( g_pVertexBuffer ) g_pVertexBuffer->Release();
       if ( g_pVertexLayout ) g_pVertexLayout->Release();
       if ( g_pEffect ) g_pEffect->Release();

       if( g_pRenderTargetView ) g_pRenderTargetView->Release();
       if( g_pSwapChain ) g_pSwapChain->Release();
       if( g_pd3dDevice ) g_pd3dDevice->Release();
}
```

Анимируем треугольник

У нас получилось вывести на экран неподвижное изображение треугольника. Сейчас давайте попытаемся модернизировать нашу программу таким образом, чтобы получить вращающийся треугольник. Задача эта совсем не так сложна, какой, возможно, кажется. Мы не зря рассматривали различные преобразования координат, и сейчас эти знания нам пригодятся. Чтобы осуществить задуманный анимационный эффект, необходимо применять преобразование вращения вокруг оси Y ко всем трем вершинам треугольника. Причем угол вращения должен увеличиваться с каждым кадром. Основное приложение должно обеспечивать подготовку всех трех матриц преобразования и их передачу в функцию вершинного шейдера. Вершинный шейдер, в свою очередь, должен обеспечивать применение преобразований к координатам вершин. Новым для нас здесь будет организация передачи данных между шейдерами и основным приложением.

Давайте сначала внесем изменения в основную программу, а потом приступим к редактированию файла эффектов. Создадим новый проект, скопируем в него исходный текст из программы, выводящей на экран треугольник (листинг 6.1). Добавим новые глобальные переменные, которые нам необходимы. А понадобится нам три матрицы, которые мы будем использовать в качестве матриц преобразования, и три указателя на интерфейс `ID3D10EffectMatrixVariable`,

который используется для связи между приложением и шейдерами (обмен данными в виде матриц). Объявление глобальных переменных сделаем так:

```
D3DXMATRIX                  g_World;
D3DXMATRIX                  g_View;
D3DXMATRIX                  g_Projection;
ID3D10EffectMatrixVariable* g_pWorldVariable = NULL;
ID3D10EffectMatrixVariable* g_pViewVariable = NULL;
ID3D10EffectMatrixVariable* g_pProjectionVariable = NULL;
```

Полностью объявление глобальных переменных программы выглядит следующим образом.

```
HWND        g_hWnd = NULL;
D3D10_DRIVER_TYPE       g_driverType = D3D10_DRIVER_TYPE_NULL;
ID3D10Device*           g_pd3dDevice = NULL;
IDXGISwapChain*         g_pSwapChain = NULL;
ID3D10RenderTargetView* g_pRenderTargetView = NULL;

ID3D10Effect*           g_pEffect = NULL;
ID3D10EffectTechnique*  g_pTechnique = NULL;
ID3D10InputLayout*      g_pVertexLayout = NULL;
ID3D10Buffer*           g_pVertexBuffer = NULL;

D3DXMATRIX                  g_World;
D3DXMATRIX                  g_View;
D3DXMATRIX                  g_Projection;
ID3D10EffectMatrixVariable* g_pWorldVariable = NULL;
ID3D10EffectMatrixVariable* g_pViewVariable = NULL;
ID3D10EffectMatrixVariable* g_pProjectionVariable = NULL;
```

Теперь мы должны предусмотреть правильную инициализацию переменных. Для этого нам потребуется добавить несколько строк в функцию `InitDirect3D10()`. Открываем эту функцию, находим строку, в которой происходит создание эффекта из файла. Меняем имя файла эффектов на "rotate.fx":

```
hr = D3DX10CreateEffectFromFile( L"rotate.fx", NULL, NULL, "fx_4_0",
          D3D10_SHADER_ENABLE_STRICTNESS, 0, g_pd3dDevice, NULL, NULL,
          &g_pEffect, NULL, NULL );
```

Пропускаем извлечение техники отображения из файла эффектов и после этой строки вставляем инициализацию связи с переменными шейдера. Выглядит она так:

```
g_pWorldVariable = g_pEffect->GetVariableByName( "World" )->AsMatrix();
g_pViewVariable = g_pEffect->GetVariableByName( "View" )->AsMatrix();
```

```
g_pProjectionVariable = g_pEffect->
                        GetVariableByName( "Projection" )->AsMatrix();
```

В качестве примера подробно разберем первую строку. Перед этим необходимо пояснить, что метод `ID3D10Effect::GetVariableByName` использует в качестве параметра имя переменной, под которым она объявлена в исходном тексте шейдера (название метода можно перевести как "получить переменную по ее имени"). У нас матричные переменные будут иметь следующие имена: `World`, `View` и `Projection`, для мировой матрицы, матрицы вида и матрицы проекции, соответственно. В первой строке происходит следующее: мы вызываем метод `ID3D10Effect::GetVariableByName`, который возвращает указатель на базовый класс переменных эффектов — `ID3D10EffectVariable`. После этого вызывается метод `ID3D10EffectVariable::AsMatrix` этого базового класса, который преобразует тип интерфейса к интерфейсу матричной переменной `ID3D10EffectMatrixVariable`, и полученное значение присваивается переменной `g_WorldVariable`. Аналогичным образом инициализируются две другие переменные.

Здесь все, что нужно, мы сделали, давайте теперь разберемся с матрицами преобразований. Проще всего вынести их инициализацию в конец функции `InitDirect3D10()`, чтобы иметь к ним быстрый доступ. Итак, добавляем новый текст после строки, в которой устанавливается тип и способ построения примитивов. Инициализируем мировую матрицу, загрузив в нее значения единичной матрицы:

```
D3DXMatrixIdentity( &g_World );
```

Далее очередь за матрицей вида:

```
D3DXVECTOR3 Eye( 0.0f, 0.0f, -1.7f );
D3DXVECTOR3 At( 0.0f, 0.0f, 0.0f );
D3DXVECTOR3 Up( 0.0f, 1.0f, 0.0f );
D3DXMatrixLookAtLH( &g_View, &Eye, &At, &Up );
```

А теперь установим матрицу проекции:

```
D3DXMatrixPerspectiveFovLH( &g_Projection, (float)D3DX_PI/4,
                            width/(float)height, 0.1f, 100.0f );
```

С инициализацией мы закончили, и полностью функция `InitDirect3D10()` должна принять следующий вид:

```
HRESULT InitDirect3D10()
{
        HRESULT hr = S_OK;

    // Размеры клиентской области окна
    RECT rc;
```

```
GetClientRect( g_hWnd, &rc );
UINT width = rc.right - rc.left;
UINT height = rc.bottom - rc.top;

// Список возможных типов устройства
D3D10_DRIVER_TYPE driverTypes[] =
{
    D3D10_DRIVER_TYPE_HARDWARE,
    D3D10_DRIVER_TYPE_REFERENCE,
};
UINT numDriverTypes = sizeof(driverTypes) / sizeof(driverTypes[0]);

// Заполняем структуру
DXGI_SWAP_CHAIN_DESC sd;
ZeroMemory( &sd, sizeof(sd) );
sd.BufferCount = 1;
sd.BufferDesc.Width = width;
sd.BufferDesc.Height = height;
sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
sd.BufferDesc.RefreshRate.Numerator = 60;
sd.BufferDesc.RefreshRate.Denominator = 1;
sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
sd.OutputWindow = g_hWnd;
sd.SampleDesc.Count = 1;
sd.SampleDesc.Quality = 0;
sd.Windowed = TRUE;

// Пытаемся создать устройство, проходя по списку
// Как только получилось - выходим из цикла
for( UINT driverTypeIndex = 0;
            driverTypeIndex < numDriverTypes; driverTypeIndex++ )
{
    g_driverType = driverTypes[driverTypeIndex];
    hr = D3D10CreateDeviceAndSwapChain( NULL,
             g_driverType, NULL, 0, D3D10_SDK_VERSION, &sd,
             &g_pSwapChain, &g_pd3dDevice );
     if( SUCCEEDED( hr ) )
          break;
}
```

```
    if( FAILED(hr) )
         return hr;

    // Представление данных для
    // буфера визуализации
    ID3D10Texture2D *pBackBuffer;
    hr = g_pSwapChain->GetBuffer( 0, __uuidof( ID3D10Texture2D ),
                                  (LPVOID*)&pBackBuffer );
    if( FAILED(hr) )
          return hr;
    // Создадим представление данных
    hr = g_pd3dDevice->CreateRenderTargetView( pBackBuffer, NULL,
                                               &g_pRenderTargetView );
    pBackBuffer->Release();
    if( FAILED(hr) )
          return hr;
    // Свяжем буфер визуализации с графическим конвейером
    g_pd3dDevice->OMSetRenderTargets( 1, &g_pRenderTargetView, NULL );

    // Настроим область отображения
    D3D10_VIEWPORT vp;
    vp.Width = width;
    vp.Height = height;
    vp.MinDepth = 0.0f;
    vp.MaxDepth = 1.0f;
    vp.TopLeftX = 0;
    vp.TopLeftY = 0;
    g_pd3dDevice->RSSetViewports( 1, &vp );

    // Используем наши шейдеры
    // в файле эффектов rotate.fx
    hr = D3DX10CreateEffectFromFile( L"rotate.fx", NULL, NULL, "fx_4_0",
             D3D10_SHADER_ENABLE_STRICTNESS, 0, g_pd3dDevice, NULL, NULL,
             &g_pEffect, NULL, NULL );

    if( FAILED( hr ) )
    {
        MessageBox( NULL, L"Не удается обнаружить файл эффектов(FX).
Файл эффектов должен находиться в той же папке, что и исполняемый
```

```
файл", L"Ошибка", MB_OK );
        return hr;
    }

    // Извлекаем технику отображения
    g_pTechnique = g_pEffect->GetTechniqueByName( "RenderWhite" );

    // Связь с переменными шейдера
    g_pWorldVariable = g_pEffect->
                         GetVariableByName( "World" )->AsMatrix();
    g_pViewVariable = g_pEffect->
                        GetVariableByName( "View" )->AsMatrix();
    g_pProjectionVariable = g_pEffect->
                        GetVariableByName( "Projection" )->AsMatrix();

    // Описываем формат входных данных
    D3D10_INPUT_ELEMENT_DESC layout[] =
    {
        { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
          D3D10_INPUT_PER_VERTEX_DATA, 0 },
    };
    UINT numElements = sizeof(layout)/sizeof(layout[0]);

    // Создаем объект входных данных
    D3D10_PASS_DESC PassDesc;
    g_pTechnique->GetPassByIndex( 0 )->GetDesc( &PassDesc );
    hr = g_pd3dDevice->CreateInputLayout( layout, numElements,
           PassDesc.pIAInputSignature, PassDesc.IAInputSignatureSize,
           &g_pVertexLayout );
    if( FAILED( hr ) )
        return hr;

    // Связываем объект входных данных с графическим конвейером
    g_pd3dDevice->IASetInputLayout( g_pVertexLayout );

    // Создаем буфер вершин
    SimpleVertex vertices[] =
    {
        D3DXVECTOR3( 0.0f, 0.5f, 0.0f ),
```

```
        D3DXVECTOR3( 0.5f, -0.5f, 0.0f ),
        D3DXVECTOR3( -0.5f, -0.5f, 0.0f ),
    };
    D3D10_BUFFER_DESC bd;
    bd.Usage = D3D10_USAGE_DEFAULT;
    bd.ByteWidth = sizeof( SimpleVertex ) * 3;
    bd.BindFlags = D3D10_BIND_VERTEX_BUFFER;
    bd.CPUAccessFlags = 0;
    bd.MiscFlags = 0;
    D3D10_SUBRESOURCE_DATA InitData;
    InitData.pSysMem = vertices;
    hr = g_pd3dDevice->CreateBuffer( &bd, &InitData, &g_pVertexBuffer );
    if( FAILED( hr ) )
        return hr;

    // Связываем буфер вершин с графическим конвейером
    UINT stride = sizeof( SimpleVertex );
    UINT offset = 0;
    g_pd3dDevice->IASetVertexBuffers( 0, 1, &g_pVertexBuffer,
                                      &stride, &offset );

    // Задаем тип и способ построения примитивов
    g_pd3dDevice->IASetPrimitiveTopology(
                        D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST );

       // Инициализируем мировую матрицу
    D3DXMatrixIdentity( &g_World );

    // Инициализируем матрицу вида
    D3DXVECTOR3 Eye( 0.0f, 0.0f, -1.7f );
    D3DXVECTOR3 At( 0.0f, 0.0f, 0.0f );
    D3DXVECTOR3 Up( 0.0f, 1.0f, 0.0f );
    D3DXMatrixLookAtLH( &g_View, &Eye, &At, &Up );

    // Инициализируем матрицу проекции
    D3DXMatrixPerspectiveFovLH( &g_Projection, (float)D3DX_PI/4,
                                width/(float)height, 0.1f, 100.0f );

    return S_OK;
}
```

А сейчас мы займемся функцией прорисовки сцены. Откроем функцию `RenderScene()`, изменения будем вносить в самое начало. Мы уже говорили, что для создания эффекта вращения нужно с каждым новым кадром увеличивать угол поворота. Давайте введем статическую переменную `t`, которая будет представлять собой величину угла поворота, зависящего от времени, прошедшего с момента запуска программы.

```
static float t = 0.0f;
```

Мы должны предусмотреть расчет этого угла для двух разных режимов работы: при использовании программной реализации графического конвейера (reference device) и при использовании его аппаратной реализации. В общем случае можно использовать такой алгоритм расчета угла:

- С помощью функции `GetTickCount()` получить количество миллисекунд, прошедших с момента включения компьютера.
- Вычислить количество миллисекунд, прошедших с первого вызова функции `RenderScene()`, то есть промежуток времени между первым и текущим кадром.
- Используя вычисленное значение, установить переменную `t`.

Программно все это можно реализовать, например, так:

```
static DWORD dwTimeStart = 0;
DWORD dwTimeCur = GetTickCount();
if( dwTimeStart == 0 )
    dwTimeStart = dwTimeCur;
t = ( dwTimeCur - dwTimeStart ) / 1000.0f;
```

В переменной `dwTimeStart` хранится количество миллисекунд, прошедшее с момента запуска системы, до первого вызова функции `RenderScene()`. Переменная `dwTimeCur` содержит количество миллисекунд от старта системы до текущего (выполняющегося в данный момент) вызова функции рисования. Их разность и даст нужное нам количество миллисекунд от первого до текущего кадра. Переменной `t` мы присваиваем значение этого промежутка времени, выраженное в секундах. Это значение мы передадим как угол поворота в радианах. Это означает, что за секунду наш треугольник будет поворачиваться примерно на 57° (на один радиан). Такой способ отлично подходит для случая, когда используется аппаратное ускорение, и кадры рисуются быстро. В случае если используется программная эмуляция, мы, конечно, тоже получим поворот треугольника, но он будет осуществляться резкими скачками. Дело в том, что в этом случае для рисования кадра требуется намного больше времени, порядка секунды. Следствием этого будет разница в угле поворота на соседних кадрах примерно в те же 57°. Обойти это можно, если увеличивать угол напрямую, без привязки ко времени.

```
t += (float)D3DX_PI * 0.0125f;
```

Так у нас с каждым последующим кадром угол будет увеличиваться примерно на 2°, мы получим более-менее плавное вращение. Реализуем оба этих подхода, применяя нужный в зависимости от типа драйвера устройства:

```
static float t = 0.0f;
if( g_driverType == D3D10_DRIVER_TYPE_REFERENCE )
{
    t += (float)D3DX_PI * 0.0125f;
}
else
{
    static DWORD dwTimeStart = 0;
    DWORD dwTimeCur = GetTickCount();
    if( dwTimeStart == 0 )
        dwTimeStart = dwTimeCur;
    t = ( dwTimeCur - dwTimeStart) / 1000.0f;
}
```

Теперь легко получить матрицу вращения, поскольку требуемый угол поворота у нас уже есть:

```
D3DXMatrixRotationY( &g_World, t );
```

После этого передадим все матрицы преобразований в соответствующие переменные шейдера, для этого применим метод `ID3D10EffectMatrixVariable::SetMatrix`. В качестве параметра передаем указатель на первый элемент матрицы:

```
g_pWorldVariable->SetMatrix( (float*)&g_World );
g_pViewVariable->SetMatrix( (float*)&g_View );
g_pProjectionVariable->SetMatrix( (float*)&g_Projection );
```

Изменения в функции рисования на этом закончены. Вскоре мы отправимся к файлу эффектов, а пока посмотрим, как должна выглядеть функция `RenderScene()` после нашей доработки:

```
void RenderScene()
{
// Угол поворота
    static float t = 0.0f;
    if( g_driverType == D3D10_DRIVER_TYPE_REFERENCE )
    {
        t += (float)D3DX_PI * 0.0125f;
    }
    else
    {
        static DWORD dwTimeStart = 0;
```

```
        DWORD dwTimeCur = GetTickCount();
        if( dwTimeStart == 0 )
            dwTimeStart = dwTimeCur;
        t = ( dwTimeCur - dwTimeStart) / 1000.0f;
    }

// Матрица поворота треугольника
// на нужный угол в зависимости от времени
    D3DXMatrixRotationY( &g_World, t );

// Передаем информацию о матрицах преобразования
    g_pWorldVariable->SetMatrix( (float*)&g_World );
    g_pViewVariable->SetMatrix( (float*)&g_View );
    g_pProjectionVariable->SetMatrix( (float*)&g_Projection );

// Очищаем вторичный буфер
//(компоненты красного, зеленого, синего, прозрачность)
    float ClearColor[4] = { 0.1f, 0.5f, 0.1f, 1.0f };
g_pd3dDevice->ClearRenderTargetView( g_pRenderTargetView,
ClearColor );

 // Рисуем наш треугольник
    D3D10_TECHNIQUE_DESC techDesc;
    g_pTechnique->GetDesc( &techDesc );
    for( UINT p = 0; p < techDesc.Passes; ++p )
    {
        g_pTechnique->GetPassByIndex( p )->Apply(0);
        g_pd3dDevice->Draw( 3, 0 );
    }

    g_pSwapChain->Present( 0, 0 );
}
```

Скопируем файл first.fx в папку с нашим новым проектом. Переименуем файл в rotate.fx и откроем его. Объявляем три глобальные переменные для матриц преобразования, при этом мы дадим им имена, о которых договорились ранее:

```
matrix World;
matrix View;
matrix Projection;
```

Значения в эти переменные мы передаем из приложения. Чтобы применить преобразования к координатам вершин, дополним функцию вершинного шейдера следующим образом:

```
PS_INPUT VS( VS_INPUT Data )
{
    PS_INPUT Out;
    Out = (PS_INPUT)0;

    // Умножаем координаты вершины
    // на матрицы преобразований
    Out.Pos = mul( Data.Pos, World );
    Out.Pos = mul( Out.Pos, View );
    Out.Pos = mul( Out.Pos, Projection );

    return Out;
}
```

Как видно из приведенного фрагмента, мы последовательно умножаем координаты вершины на каждую из матриц преобразования. Сохраняем изменения в файл эффектов. Пробуем откомпилировать и запустить проект. Если все изменения внесены правильно, проект должен запуститься без всяких проблем. Можно спокойно понаблюдать за вращающимся треугольником, наслаждаясь результатами своего труда. Однако когда треугольник повернется на девяносто градусов, мы, вместо его обратной стороны, увидим только пустое окно. Треугольник исчез! Такого мы не программировали. Что же вдруг случилось? Исчезновение связано с тем, что по умолчанию включено отсечение невидимых граней. Считается, что у треугольника видима только одна из его сторон (граней), и это, в общем-то, правильно. Треугольники, как правило, образуют какой-то замкнутый объемный объект, и грани, обращенные в сторону от камеры, видны не будут, и их можно отбросить, сразу исключить из визуализации. Видимость грани определяется углом между нормалью к ее поверхности и направлением, в котором сориентирована камера. Если этот угол больше девяноста градусов — грань невидима. Нам пока не важно, каким образом вычисляется нормаль к поверхности треугольника, нам сейчас интересно, как отключить отсечение невидимых граней, чтобы при вращении треугольника наблюдать обе его грани.

Здесь мы вкратце познакомимся с установкой режимов (state) растеризатора. Параметры этих режимов определяют, как отрисованные треугольники взаимодействуют с буфером визуализации, друг с другом, а также с некоторыми

другими объектами. Для того чтобы выключить отсечение невидимых граней, достаточно определить специальный режим работы растеризатора. Введем его описание перед описанием техники отображения:

```
RasterizerState rsNoCulling { CullMode = None; };
```

В самой технике последней строкой добавим установку описанного режима:

```
SetRasterizerState( rsNoCulling );
```

Теперь, если запустить проект, мы будем видеть, как треугольник, вращаясь, по очереди показывает нам то одну, то другую грань. Один кадр из получившейся анимации показан на рис. 6.4.

Полностью исходный текст файла эффектов rotate.fx приведен в листинге 6.2, сам этот файл и проект для отображения на экране вращающегося треугольника можно найти на компакт-диске в директории Glava6\Rotate. Можно попробовать, пользуясь материалом из *гл. 2*, добавить возможность переключения в полноэкранный режим и обратно. Попытайтесь самостоятельно доработать программу, в случае затруднений обратитесь к готовому проекту, где эта возможность уже реализована. Проект находится на компакт-диске в директории Glava6\RotateFullScr.

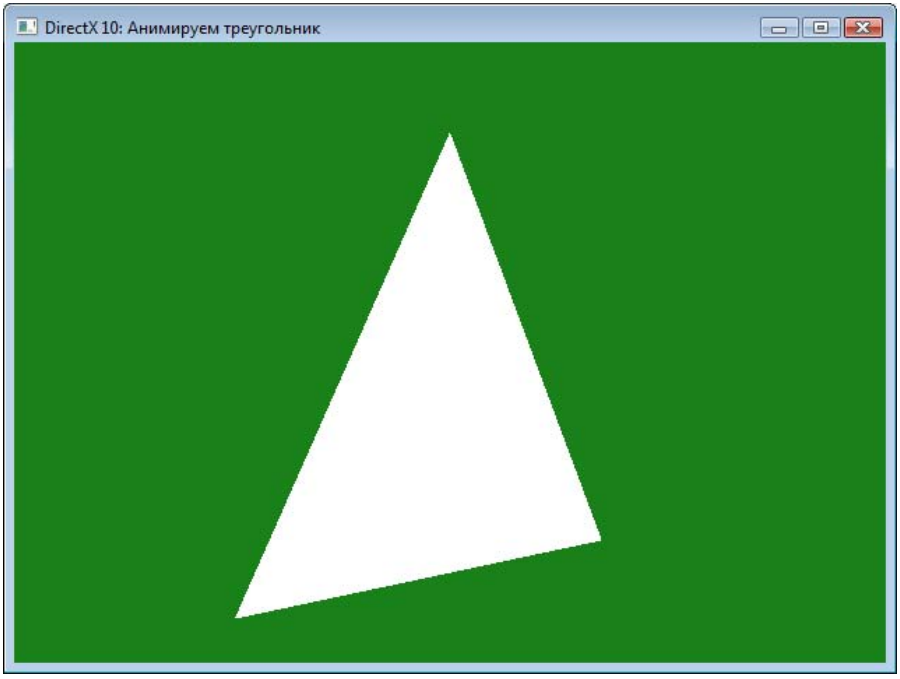

Рис. 6.4. Вращающийся треугольник

Листинг 6.2

```
// Анимация треугольника
// Файл rotate.fx
//--------------------------------------------------------------------------
// Структуры данных
//--------------------------------------------------------------------------
struct VS_INPUT
{
    float4 Pos : POSITION;
};

struct PS_INPUT
{
    float4 Pos : SV_POSITION;
};

//--------------------------------------------------------------------------
// Глобальные переменные
//--------------------------------------------------------------------------
matrix World;
matrix View;
matrix Projection;

//--------------------------------------------------------------------------
// Функция вершинного шейдера
//--------------------------------------------------------------------------
PS_INPUT VS( VS_INPUT Data )
{
    PS_INPUT Out;
    Out = (PS_INPUT)0;

    // Умножаем координаты вершины
    // на матрицы преобразований
    Out.Pos = mul( Data.Pos, World );
    Out.Pos = mul( Out.Pos, View );
    Out.Pos = mul( Out.Pos, Projection );

    return Out;
}
```

```
//--------------------------------------------------------------------------
// Функция пиксельного шейдера
//--------------------------------------------------------------------------
float4 PS( PS_INPUT Pos ) : SV_Target
{
    return float4( 1.0f, 1.0f, 1.0f, 1.0f );// Белый цвет
}
//--------------------------------------------------------------------------
// Техника отображения
//--------------------------------------------------------------------------
        // Создаем набор настроек растеризатора
        // без отсечения невидимых граней
        RasterizerState rsNoCulling { CullMode = None; };
technique10 RenderWhite
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, VS() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, PS() ) );
        // Невидимые грани не отсекать
        SetRasterizerState( rsNoCulling );
    }
}
```

Больше цвета!

Мы очень ощутимо продвинулись вперед, уже заставляем двигаться объекты виртуального мира. Тем не менее, почему-то до сих пор рисуем все одним белым цветом. Давайте выясним, как разнообразить цветовую гамму.

Мы знаем, что за непосредственный вывод информации о пикселах изображения отвечает пиксельный шейдер. Наш пиксельный шейдер пока что всегда возвращал белый цвет пиксела. Как это изменить? Нужно передавать информацию о цвете вершины вместе с ее координатами и использовать ее с помощью пиксельного шейдера. Для этого нам, в очередной раз, предстоит провести несколько изменений. Создаем новый проект и копируем в него исходный текст из проекта с вращающимся треугольником. В первую очередь, мы вносим дополнения в структуру, которая описывает формат вершины:

```
struct SimpleVertex
{
```

```
    D3DXVECTOR3 Pos;
    D3DXVECTOR4 Color;
};
```

После этого мы откроем функцию `InitDirect3D10()` и поправим формат входных данных таким образом, чтобы он согласовывался с новым, расширенным форматом вершины. Потребуется ввести второй элемент в массив описания формата:

```
  D3D10_INPUT_ELEMENT_DESC layout[] =
    {
        { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
          D3D10_INPUT_PER_VERTEX_DATA, 0 },
        { "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12,
          D3D10_INPUT_PER_VERTEX_DATA, 0 }
    };
```

Давайте проговорим, какую информацию несет этот второй элемент. Семантическое имя — "COLOR", индекс семантики не используется. Элемент представляет собой четыре тридцатидвухразрядных значения с плавающей точкой. Элемент вводится через нулевой входной канал, смещение данных элемента от первого байта данных составляет 12 байт (предыдущий элемент содержит три значения, каждое из которых кодируется четырьмя байтами, $3 \times 4 = 12$ байт). Данные, содержащиеся в элементе, относятся к одной вершине, создание экземпляров не производится.

Формат данных мы определили, давайте дополним и сами данные о вершине, скорректируем массив `vertices[]`. Для определенности давайте вершинам треугольника установим их цвета как красный, зеленый и синий соответственно:

```
SimpleVertex vertices[] =
{
    D3DXVECTOR3( 0.0f, 0.5f, 0.0f ),
    D3DXVECTOR4 ( 1.0f, 0.0f, 0.0f, 1.0f ),
    D3DXVECTOR3( 0.5f, -0.5f, 0.0f ),
    D3DXVECTOR4 ( 0.0f, 1.0f, 0.0f, 1.0f ),
    D3DXVECTOR3( -0.5f, -0.5f, 0.0f ),
    D3DXVECTOR4 ( 0.0f, 0.0f, 1.0f, 1.0f )
};
```

Вот и все изменения в исходном тексте приложения. Остальные исправления нам предстоит сделать в файле эффектов. Скопируем файл rotate.fx из предыдущего проекта и переименуем его в color.fx. Да, чуть не забыли, пока мы еще не далеко ушли от функции `InitDirect3D10()`, исправим имя файла эффектов в вызове функции `D3DX10CreateEffectFromFile()` на "color.fx"

и имя извлекаемой техники отображения — на "RenderColor" (пока ее в файле нет, но мы очень скоро это исправим).

Открываем файл эффектов. Так как мы изменили формат входных данных в основной программе, необходимо внести изменения и в структуры входных параметров шейдеров. (В противном случае дополнительная информация просто потеряется). Добавить нужно еще один параметр, сохраняя тип данных и семантику, которые были заданы в описании входных данных:

```
struct VS_INPUT
{
    float4 Pos : POSITION;
    float4 Color : COLOR;
};

struct PS_INPUT
{
    float4 Pos : SV_POSITION;
    float4 Color : COLOR;
};
```

Итак, информация о вершине поступает на вход вершинного шейдера. К ее координатам применяются соответствующие преобразования, а что делать с цветом? Цвет вершины мы просто передаем на выход без изменений, сделать это нужно обязательно, в противном случае пиксельный шейдер эту информацию не получит. Обновленная функция вершинного шейдера должна выглядеть так:

```
PS_INPUT VS( VS_INPUT Data )
{
    PS_INPUT Out;
    Out = (PS_INPUT)0;

    // Умножаем координаты вершины
    // на матрицы преобразований
    Out.Pos = mul (Data.Pos, World);
    Out.Pos = mul (Out.Pos, View);
    Out.Pos = mul (Out.Pos, Projection);

    // Передаем цвет без изменений
    Out.Color = Data.Color;

    return Out;
}
```

Для пиксельного шейдера, выводящего пикселы в соответствии с полученными цветами вершин, напишем отдельную функцию:

```
float4 PS_Color( PS_INPUT input ) : SV_Target
{
    return input.Color;
}
```

Как можно заметить, эта функция тоже возвращает цвет из данных, полученных на входе. Старую функцию, которая возвращает белый цвет, оставим пока на месте. Опишем дополнительную технику отображения, в которой используем новый пиксельный шейдер:

```
technique10 RenderColor
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, VS() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, PS_Color() ) );
        // Невидимые грани не отсекать
        SetRasterizerState( rsNoCulling );
    }
}
```

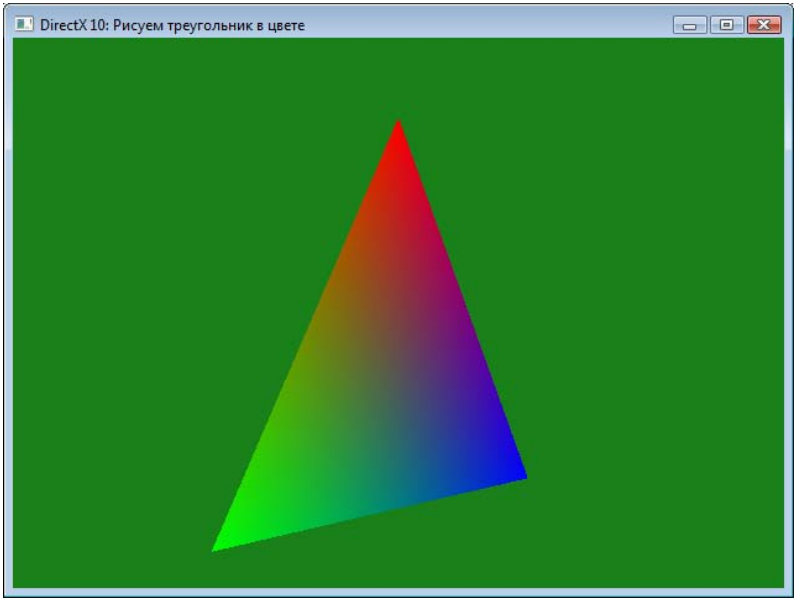

Рис. 6.5. Вращающийся треугольник в цвете

Все, новый файл эффектов готов, сохраняем его. Исходный текст файла эффектов приведен в листинге 6.3, готовый проект можно найти на прилагаемом компакт-диске в директории Glava6\Color. Скомпилируем и запустим проект. В окне приложения должен появиться вращающийся треугольник с плавными цветовыми переходами между всеми вершинами (рис. 6.5).

Листинг 6.3

```
// Анимированный треугольник в цвете
// Файл color.fx
//-----------------------------------------------------------------------
// Структуры данных
//-----------------------------------------------------------------------
struct VS_INPUT
{
    float4 Pos : POSITION;
    float4 Color: COLOR;
};

struct PS_INPUT
{
    float4 Pos : SV_POSITION;
    float4 Color: COLOR;
};

//-----------------------------------------------------------------------
// Глобальные переменные
//-----------------------------------------------------------------------
matrix World;
matrix View;
matrix Projection;

//-----------------------------------------------------------------------
// Функция вершинного шейдера
//-----------------------------------------------------------------------
PS_INPUT VS( VS_INPUT Data )
{
    PS_INPUT Out;
    Out = (PS_INPUT)0;
```

```
    // Умножаем координаты вершины
    // на матрицы преобразований
    Out.Pos = mul (Data.Pos, World);
    Out.Pos = mul (Out.Pos, View);
    Out.Pos = mul (Out.Pos, Projection);

    // Передаем цвет без изменений
    Out.Color = Data.Color;

    return Out;
}
//--------------------------------------------------------------------------
// Функция пиксельного шейдера (возвращает всегда белый цвет)
//--------------------------------------------------------------------------
float4 PS( PS_INPUT Pos ) : SV_Target
{
    return float4( 1.0f, 1.0f, 1.0f, 1.0f );// Белый цвет
}

//--------------------------------------------------------------------------
// Функция пиксельного шейдера
//--------------------------------------------------------------------------
float4 PS_Color( PS_INPUT input ) : SV_Target
{
    return input.Color;
}

//--------------------------------------------------------------------------
// Техника отображения
//--------------------------------------------------------------------------
     // Создаем набор настроек растеризатора
        // без отсечения невидимых граней
        RasterizerState rsNoCulling { CullMode = None; };

technique10 RenderWhite
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, VS() ) );
```

```
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, PS() ) );
        // Невидимые грани не отсекать
        SetRasterizerState( rsNoCulling );
    }
}

technique10 RenderColor
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, VS() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, PS_Color() ) );
        // Невидимые грани не отсекать
        SetRasterizerState( rsNoCulling );
    }
}
```

Проделав все эти эксперименты, мы увидели, какую важную роль играют шейдеры в Direct3D 10. Мы ведь смогли добиться значительных изменений выводимого изображения в основном за счет изменений шейдеров, основную программу мы корректировали незначительно. В следующей главе нам предстоит возложить на шейдеры еще большую нагрузку, такую как расчет освещения, но обо всем по порядку…

Глава 7

Больше реализма: освещение, материалы, текстуры

Пока все то, что мы выводили на экран, выглядит одинаково, как будто сделанным из пластика, только цветной треугольник немного оживляет картину. Однако трехмерная графика, наверное, не завоевала бы такую популярность, если бы она не давала возможность с высокой точностью имитировать окружающий нас мир, позволяя погружаться в мир придуманный, например, в мир компьютерной игры. Чтобы добиться реалистичности изображения, необходимо уметь моделировать не только поверхности физических тел, но и другие явления окружающего мира, с которыми мы постоянно имеем дело. Начать, наверное, следует с моделирования освещения.

Как включить свет?

Такая постановка вопроса, скорее всего, вас удивит. В реальном мире свет мы включаем для того, чтобы можно было лучше рассмотреть предметы, без этого скрытые в темноте. В программах, которые мы писали, все объекты хорошо видны и темных участков нигде нет. Зачем же еще придумывать какое-то освещение? Без использования источников света объекты выглядят неестественно, как будто бы освещенными светом одинаковой интенсивности со всех сторон. Вообще говоря, таким образом мы задавали *общее освещение* (ambient light). Общее освещение — свет, не исходящий из какого-то конкретного источника, который как бы "размазан" по всему объему трехмерной сцены. Он дает всем поверхностям постоянное освещение, не зависящее от их ориентации. В реальном мире, чтобы получить такой эффект, нужно очень постараться. Для исправления этой ситуации необходимо доработать пиксельный шейдер так, чтобы интенсивность цвета каждого пиксела рассчитывалась на основе переданных из основной программы информации об источниках света (положении в пространстве, ориентации, интенсивности, цвете

излучаемого света). Перед тем как мы познакомимся с одной из возможных реализаций модели освещения, давайте поподробнее рассмотрим, какие существуют типы источников света и чем они друг от друга отличаются. Мы сконцентрируем внимание на тех трех типах, которые поддерживались в девятой версии Direct3D.

Точечный источник света

Точечный источник света представляет собой "светящуюся точку", равномерно испускающую свет во всех направлениях (рис. 7.1).

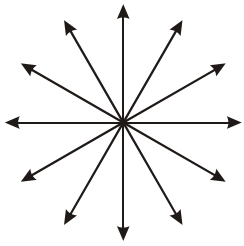

Рис. 7.1. Точечный источник света

Характеризуется он положением в пространстве и цветом излучаемого света. Примером источника такого типа можно считать обычную электрическую лампочку. Хотя, конечно, настоящий точечный источник геометрических размеров не имеет.

Направленный свет

Этот источник света характеризуется только направлением и цветом излучаемого света (рис. 7.2).

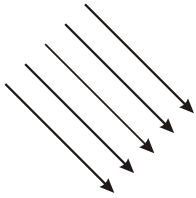

Рис. 7.2. Направленный свет

Все испускаемые лучи параллельны между собой. Направленный источник света можно представить как точечный источник света, находящийся

на бесконечно большом расстоянии. В качестве примера можно привести солнечный свет. Лучи Солнца, достигающие поверхности Земли, конечно, не идеально параллельны, но максимальный угол между ними (между лучом на Северном полюсе и лучом на Южном полюсе Земли) составляет менее 0,005°.

Прожектор

Из названия уже понятно, что этот источник собой представляет. Он характеризуется положением в пространстве, цветом и направлением, в котором излучается свет. Сам излучаемый свет подразделяется на две зоны: яркий внутренний конус и наружный конус, в котором свет постепенно затухает (рис. 7.3).

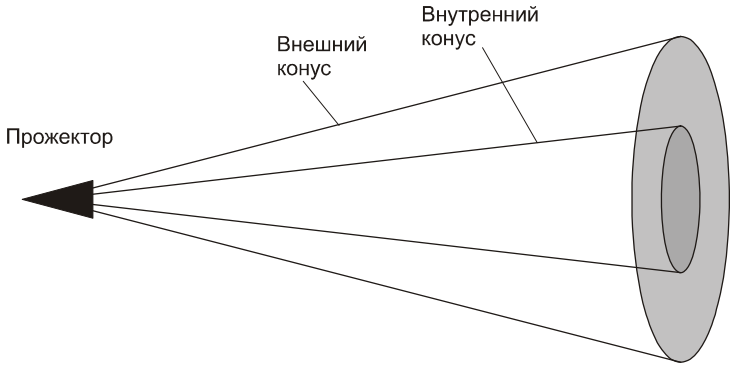

Рис. 7.3. Прожектор: внутренний и внешний конусы света

Ну вот, в общих чертах понятно, кто есть кто, вернее, — что есть что. К сожалению, для того, чтобы использовать эти источники света в своих шейдерах, этой информации не хватит. Нам понадобятся данные о том, как изменяется интенсивность света с удалением от его источника.

Математика освещенности

Начнем немного не по порядку, с самого простого случая. Этим простым случаем является направленный свет, так как его интенсивность с увеличением расстояния не изменяется. Формула показывает, что интенсивность света, дошедшего до объекта (I_o), равна интенсивности самого источника света (I_u):

$$I_o = I_u .$$

У точечного источника света интенсивность убывает обратно пропорционально квадрату расстояния, эту зависимость можно выразить таким образом:

$$I_o = \frac{I_u}{d^2}.$$

Для того чтобы определить интенсивность света от прожектора, нужно иметь в виду, что общая интенсивность света здесь зависит не только от расстояния от источника света до точки, куда пришел луч, но и от того, в каком световом конусе эта точка находится (рис. 7.4).

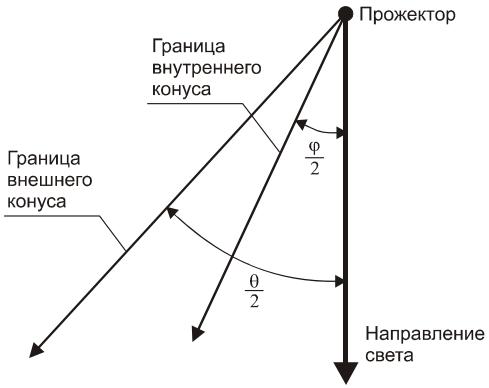

Рис. 7.4. Световые конусы прожектора

Другими словами, кроме расстояния нам также необходимо знать угол между направлением света и вектором от источника света до точки на поверхности (рис. 7.5).

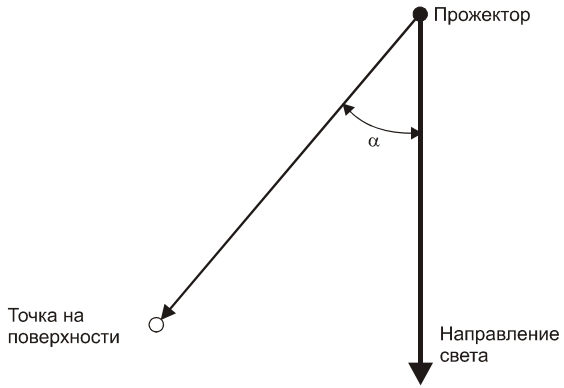

Рис. 7.5. Угол для расчета интенсивности света

Интенсивность света будет зависеть от этого угла таким образом: если $\alpha < \frac{\varphi}{2}$ (точка находится во внутреннем конусе), то $I_y = 1.0$, если $\alpha > \frac{\theta}{2}$ (точка не попадает во внешний конус), то $I_y = 0.0$. В любом другом случае (точка находится во внутреннем конусе) интенсивность света определяется выражением

$$I_y = \left(\frac{Cos(\alpha) - Cos\left(\frac{\varphi}{2}\right)}{Cos\left(\frac{\theta}{2}\right) - Cos\left(\frac{\varphi}{2}\right)} \right)^F,$$

где F представляет собой коэффициент затухания света при переходе от границы внутреннего конуса к границе внешнего. Для снижения объема вычислений значение этого коэффициента можно принять равным единице.

Свет от прожектора также ослабляется с увеличением расстояния от источника. В принципе, можно использовать формулу для точечного источника, но, чтобы иметь больше возможностей для регулирования, применяют несколько иной подход. Интенсивность света прожектора в зависимости от расстояния определяют следующим образом:

$$I_p = \frac{1}{Ad^2 + Bd + C},$$

где

- A — значение, определяющее скорость затухания света в зависимости от квадрата расстояния;
- B — значение, определяющее линейную зависимость от расстояния; в некоторых случаях для получения более плавной кривой затухания, это значение устанавливают больше значения A;
- C — значение, определяющее общее смещение кривой.

Обратите внимание, что при значениях $A = 1$, $B = 0$ и $C = 0$ функция превращается в обычную обратную квадратичную зависимость, как в случае точечного источника света. Итоговая интенсивность света от прожектора определяется следующей формулой: $I_o = I_y \cdot I_p$.

Модель освещения

Тот факт, что мы можем видеть предметы вокруг себя, объясняется тем, что предметы не поглощают весь падающий на них свет, а отражают какую-то его часть. Разные предметы отражают свет по-разному: например, зеркало

отражает практически весь падающий свет в каком-то одном направлении, а кусок бархата просто равномерно рассеивает свет. Для описания того, как поверхности разного рода отражают, поглощают, преломляют свет, существует множество разнообразных моделей освещения. Так как мы с вами находимся лишь в самом начале на пути постижения искусства программирования шейдеров, мы рассмотрим самую простую модель — диффузное освещение. Основная идея, заложенная в модель, заключается в том, что все поверхности одинаково отражают свет во всех направлениях, количество отраженного света зависит от угла падения луча и подчиняется закону косинусов Ламберта. Закон косинусов Ламберта (рис. 7.6) выражается следующей формулой: $I = Cos(\theta)$, или $I = \vec{N} \cdot \vec{L}$, здесь $\vec{N}$ представляет собой вектор нормали к поверхности, а $\vec{L}$ — это вектор, направленный из точки, в которой рассчитывается освещенность, в точку, где находится источник света.

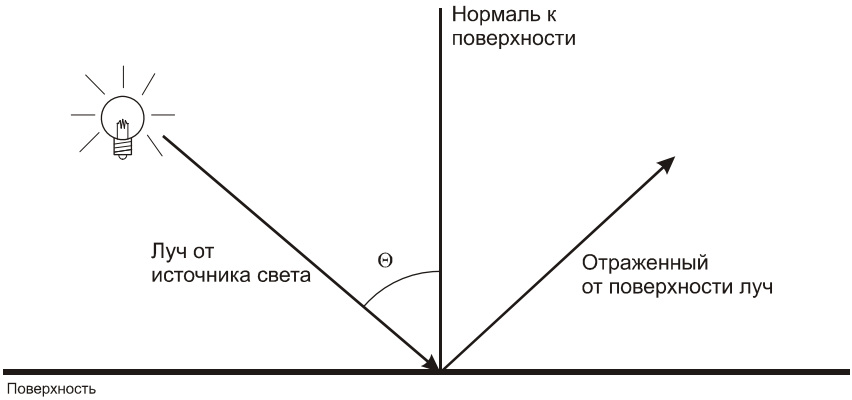

Рис. 7.6. Закон косинусов Ламберта

Согласно этому закону, количество отраженного света уменьшается с увеличением угла между лучом света, падающим на поверхность, и нормалью к этой поверхности. Попробуем для себя как-то объяснить эту зависимость, в этом нам поможет рис. 7.7.

В верхней части рисунка поверхность перпендикулярна световым лучам и от нее отражается максимально возможное количество света. В нижней части рисунка поверхность находится под углом к лучам света. Не нужно никакой математики, чтобы увидеть, что в этом случае количество света, отраженное от поверхности, заметно меньше.

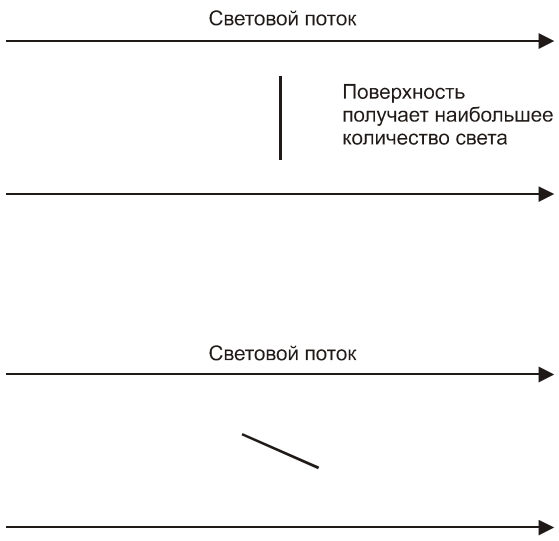

Рис. 7.7. Зависимость освещенности поверхности от угла падения лучей света

Итак, собрав все вместе, мы можем записать уравнение для вычисления освещенности любой точки на поверхности. Итоговое уравнение визуализации для диффузной модели освещения записывается следующим образом:

$$I_o = D \cdot I_u \cdot Cos(\theta),$$

где D — цвет поверхности, на которую падает луч;

I_u — интенсивность излучения падающего света;

θ — угол между нормалью к поверхности и вектором с началом в данной точке поверхности, направленным на источник света.

Теперь мы готовы к реализации модели освещения с помощью шейдеров. Давайте попробуем перечислить, какие изменения нам предстоит внести в программу. В первую очередь, наверное, в голову приходит необходимость ввести в шейдер переменные, описывающие свойства источника света. Иначе, что же это за освещение без света? Также нам понадобятся координаты нормали к каждой вершине, необходимо добавить еще один элемент в формат вершины.

В качестве отправной точки для внесения изменений, мы могли бы, конечно, использовать и программу, рисующую треугольник. Но для того, чтобы иметь возможность оценить эффект от того или иного источника света, одного треугольника будет явно не достаточно. Поэтому для реализации модели освещения мы создадим программу, выводящую на экран вращающийся куб, и заодно познакомимся с индексным рисованием примитивов.

Полигон для испытаний

Создадим новый проект, скопируем в него исходный текст из программы, которая выводит вращающийся треугольник (*см. листинг 6.1*). В папке с проектом также нужно создать файл с именем "DirectLight.fx", в нем мы впоследствии сохраним шейдеры, производящие расчет освещения.

Перед тем как мы перейдем непосредственно к редактированию исходного текста, посмотрим, что собой представляет индексное рисование примитивов, которое мы решили использовать. Чем плох способ, которым мы пользовались до сих пор, просто последовательно задавая координаты вершин? Для описания треугольника такой метод действительно подходит идеально. Рассмотрим этот же метод применительно к кубу. Куб содержит восемь вершин и шесть граней (рис. 7.8).

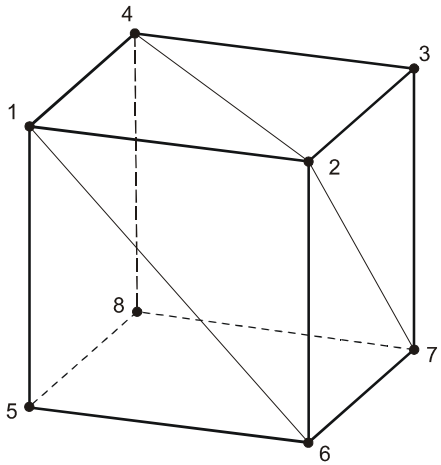

Рис. 7.8. Вершины куба

Каждая грань, в свою очередь, состоит из двух треугольников. Таким образом, используя прежний метод, всего нам нужно описать вершины двенадцати треугольников. Для этого нам потребуется ввести координаты $12 \times 3 = 36$ вершин. И это при том, что у многих вершин из этого количества координаты будут повторяться: ведь вершин у куба всего восемь. Очевидно, что имеет место не очень рациональное использование памяти, хотя сам метод вполне работоспособен. Вот если бы можно было как-то указать программе, что для первого треугольника нужно взять из буфера вершины с номерами 1, 2, 3, для следующего треугольника — вершины с номерами 1, 3, 4 и т. д… Именно так

и работает индексное рисование примитивов (в нашем случае примитивы — треугольники). Создается специальный индексный буфер, который содержит последовательность номеров элементов (индексы) из буфера вершин, из которых нужно составить примитивы. Создание и работа с индексным буфером полностью аналогична работе с буфером вершин. Единственным отличием, пожалуй, будет вызов для рисования примитивов метода `ID3D10Device::DrawIndexed`, но об этом ниже.

Начнем работу над программой. Откорректируем формат вершины таким образом, чтобы он содержал кроме координат самой вершины еще и координаты связанной с ней нормали. Новый формат представляет собой следующую структуру:

```
struct SimpleVertex
{
       D3DXVECTOR3 Pos;
       D3DXVECTOR3 Normal;
};
```

Теперь структура содержит два трехкомпонентных вектора: один для хранения координат вершины в пространстве, второй — для хранения координат вектора нормали, связанной с ней. Перейдем к глобальным переменным. Добавим указатель на интерфейс индексного буфера:

```
ID3D10Buffer* g_pIndexBuffer = NULL;
```

Здесь же нам нужно объявить переменные для шейдера, которые отвечают за описание источника света. Условимся, что источник света у нас будет один, и он пока что будет простейшего для реализации типа — направленный свет. Источник света такого типа, как мы рассмотрели выше, характеризуется двумя параметрами: направлением лучей и их цветом. Объявим соответствующие переменные для передачи шейдеру:

```
ID3D10EffectVectorVariable* g_pLightColor = NULL;
ID3D10EffectVectorVariable* g_pLightDirection = NULL;
```

Объявление глобальных переменных должно принять следующий вид:

```
HWND        g_hWnd = NULL;
D3D10_DRIVER_TYPE       g_driverType = D3D10_DRIVER_TYPE_NULL;
ID3D10Device*           g_pd3dDevice = NULL;
IDXGISwapChain*         g_pSwapChain = NULL;
ID3D10RenderTargetView* g_pRenderTargetView = NULL;

ID3D10Effect*           g_pEffect = NULL;
ID3D10EffectTechnique*  g_pTechnique = NULL;
```

```
ID3D10InputLayout*      g_pVertexLayout = NULL;
ID3D10Buffer*           g_pVertexBuffer = NULL;

ID3D10Buffer*               g_pIndexBuffer = NULL;

ID3D10EffectMatrixVariable* g_pWorldVariable = NULL;
ID3D10EffectMatrixVariable* g_pViewVariable = NULL;
ID3D10EffectMatrixVariable* g_pProjectionVariable = NULL;
ID3D10EffectVectorVariable* g_pLightColor = NULL;
ID3D10EffectVectorVariable* g_pLightDirection = NULL;
D3DXMATRIX                  g_World;
D3DXMATRIX                  g_View;
D3DXMATRIX                  g_Projection;
```

Здесь мы закончили, больше никаких изменений нет. Переходим к функции `InitDirect3D10()`. Находим строку, где происходит вызов функции создания эффекта из файла, и меняем имя файла эффектов на "DirectLight.fx". Немного ниже меняем имя извлекаемой техники отображения на "RenderDirectLight". С косметическими исправлениями все. Теперь установим связь между переменными, хранящими параметры источника света, и соответствующими переменными шейдера:

```
g_pLightColor = g_pEffect->
                    GetVariableByName( "LightColor" )->AsVector();
g_pLightDirection = g_pEffect->
                    GetVariableByName( "LightDirection" )->AsVector();
```

Исправим формат входных данных для процессора видеокарты таким образом, чтобы он соответствовал новому формату вершины. Для этого второй элемент массива должен иметь семантическое имя `NORMAL` и формат данных такой же, как у первого элемента массива:

```
D3D10_INPUT_ELEMENT_DESC layout[] =
{
  { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
        D3D10_INPUT_PER_VERTEX_DATA, 0 },
  { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
        D3D10_INPUT_PER_VERTEX_DATA, 0 }
};
```

Сейчас мы должны заполнить массив `vertices[]` новыми координатами вершин. И вот вопрос: как вы думаете, сколько вершин нам нужно занести в массив? Восемь? Так бы оно и было, если бы мы хотели нарисовать один лишь куб, без моделирования освещения. Но нам к каждой вершине нужно

приложить координаты нормали, а это многое меняет. Обратимся к рис. 7.9, на котором изображен куб с вершинами и соответствующими им нормалями.

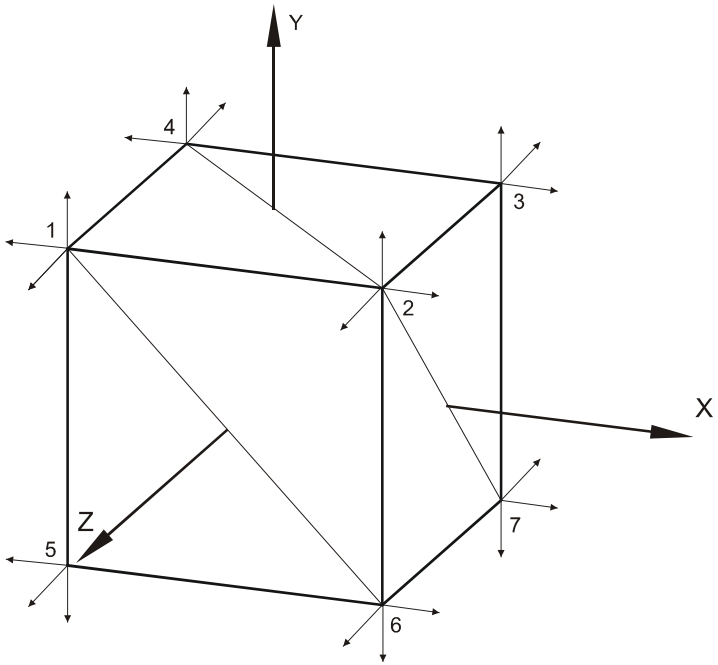

Рис. 7.9. Нормали вершин куба

И что же мы видим? Из каждой вершины выходит по три нормали. Как это понимать? Мы ведь в формате вершины определили только одну нормаль на вершину, да и не вписывается такое количество нормалей в нашу модель освещения. Одним словом — нормаль у вершины должна быть одна. Между тем, объясняется все очень просто. Мы уже упоминали, что вершины куба участвуют в образовании нескольких граней, точнее говоря, каждая вершина сопряжена с тремя гранями. Вот и получается, что, например, вершина "1", "участвуя" в треугольнике верхней грани, должна иметь нормаль, направленную вертикально вверх. Та же самая вершина, "участвуя" в треугольнике, образующем половину левой грани, уже должна иметь нормаль, направленную влево. То же самое справедливо для случая с гранью, направленной прямо на нас. Какой из этого вывод? Для правильного формирования граней нам придется записать вершину "1" в массив `vertices[]` три раза, для каждой грани в отдельности. Каждый раз координаты в пространстве будут одни и те же, элементы массива будут отличаться только координатами нормалей. Давайте для простоты будем

считать, что мы строим куб, у которого каждая вершина отстоит от начала координат на единицу по всем координатам. Например, у точки "1" координаты будут такие: (-1.0, 1.0, -1.0). Чтобы не запутаться, будем записывать координаты вершин в массив поочередно для каждой грани куба. Вот так, например, наша запись будет выглядеть для верхней грани:

```
{D3DXVECTOR3(-1.0f, 1.0f, -1.0f), D3DXVECTOR3(0.0f, 1.0f, 0.0f)}, // 1
{D3DXVECTOR3( 1.0f, 1.0f, -1.0f), D3DXVECTOR3(0.0f, 1.0f, 0.0f)}, // 2
{D3DXVECTOR3( 1.0f, 1.0f,  1.0f), D3DXVECTOR3(0.0f, 1.0f, 0.0f)}, // 3
{D3DXVECTOR3(-1.0f, 1.0f,  1.0f), D3DXVECTOR3(0.0f, 1.0f, 0.0f)}  // 4
```

Обратите внимание, как задается нормаль для верхней грани: единичный вектор, направленный в положительном направлении оси Y. Аналогичным образом заполняется массив и для остальных граней куба. Вот так выглядит полностью заполненный массив:

```
SimpleVertex vertices[] =
{
{D3DXVECTOR3(-1.0f, 1.0f,-1.0f ), D3DXVECTOR3( 0.0f, 1.0f, 0.0f )},
{D3DXVECTOR3( 1.0f, 1.0f,-1.0f ), D3DXVECTOR3( 0.0f, 1.0f, 0.0f )},
{D3DXVECTOR3( 1.0f, 1.0f, 1.0f ), D3DXVECTOR3( 0.0f, 1.0f, 0.0f )},
{D3DXVECTOR3(-1.0f, 1.0f, 1.0f ), D3DXVECTOR3( 0.0f, 1.0f, 0.0f )},

{D3DXVECTOR3(-1.0f,-1.0f,-1.0f ), D3DXVECTOR3( 0.0f, -1.0f, 0.0f )},
{D3DXVECTOR3( 1.0f,-1.0f,-1.0f ), D3DXVECTOR3( 0.0f, -1.0f, 0.0f )},
{D3DXVECTOR3( 1.0f,-1.0f, 1.0f ), D3DXVECTOR3( 0.0f, -1.0f, 0.0f )},
{D3DXVECTOR3(-1.0f,-1.0f, 1.0f ), D3DXVECTOR3( 0.0f, -1.0f, 0.0f )},

{D3DXVECTOR3(-1.0f,-1.0f, 1.0f ), D3DXVECTOR3(-1.0f, 0.0f, 0.0f )},
{D3DXVECTOR3(-1.0f,-1.0f,-1.0f ), D3DXVECTOR3(-1.0f, 0.0f, 0.0f )},
{D3DXVECTOR3(-1.0f, 1.0f,-1.0f ), D3DXVECTOR3(-1.0f, 0.0f, 0.0f )},
{D3DXVECTOR3(-1.0f, 1.0f, 1.0f ), D3DXVECTOR3(-1.0f, 0.0f, 0.0f )},

{D3DXVECTOR3( 1.0f,-1.0f, 1.0f ), D3DXVECTOR3( 1.0f, 0.0f, 0.0f )},
{D3DXVECTOR3( 1.0f,-1.0f,-1.0f ), D3DXVECTOR3( 1.0f, 0.0f, 0.0f )},
{D3DXVECTOR3( 1.0f, 1.0f,-1.0f ), D3DXVECTOR3( 1.0f, 0.0f, 0.0f )},
{D3DXVECTOR3( 1.0f, 1.0f, 1.0f ), D3DXVECTOR3( 1.0f, 0.0f, 0.0f )},

{D3DXVECTOR3(-1.0f,-1.0f,-1.0f ), D3DXVECTOR3( 0.0f, 0.0f,-1.0f )},
{D3DXVECTOR3( 1.0f,-1.0f,-1.0f ), D3DXVECTOR3( 0.0f, 0.0f,-1.0f )},
{D3DXVECTOR3( 1.0f, 1.0f,-1.0f ), D3DXVECTOR3( 0.0f, 0.0f,-1.0f )},
{D3DXVECTOR3(-1.0f, 1.0f,-1.0f ), D3DXVECTOR3( 0.0f, 0.0f,-1.0f )},
```

```
{D3DXVECTOR3(-1.0f,-1.0f, 1.0f ), D3DXVECTOR3( 0.0f, 0.0f, 1.0f )},
{D3DXVECTOR3( 1.0f,-1.0f, 1.0f ), D3DXVECTOR3( 0.0f, 0.0f, 1.0f )},
{D3DXVECTOR3( 1.0f, 1.0f, 1.0f ), D3DXVECTOR3( 0.0f, 0.0f, 1.0f )},
{D3DXVECTOR3(-1.0f, 1.0f, 1.0f ), D3DXVECTOR3( 0.0f, 0.0f, 1.0f )}
};
```

Замечание

Может сложиться неверное впечатление, что нормаль вершины всегда совпадает с нормалью к поверхности грани. Для куба они действительно совпадают, но для более сложных геометрических форм этого совпадения может и не быть (рис. 7.10).

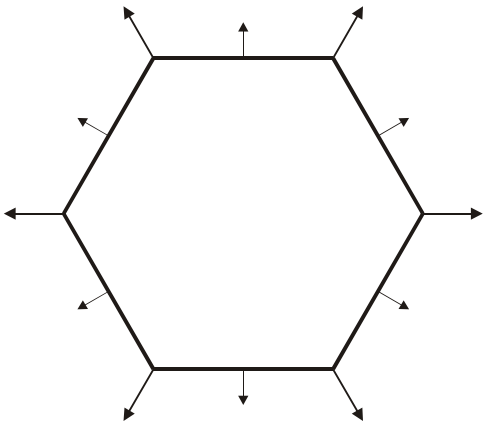

Рис. 7.10. Пример случая, когда нормали вершин не совпадают с нормалями к поверхности

Заполним структуру, описывающую буфер вершин:

```
D3D10_BUFFER_DESC bd;
bd.Usage = D3D10_USAGE_DEFAULT;
bd.ByteWidth = sizeof( SimpleVertex ) * 24;// 24 вершины с нормалями
bd.BindFlags = D3D10_BIND_VERTEX_BUFFER;
bd.CPUAccessFlags = 0;
bd.MiscFlags = 0;
```

Здесь у нас изменилось только количество байт, выделяемое для буфера: как обычно, размер данных одной вершины в байтах умножается на их общее количество — 24 (8 вершин, каждая входит в массив по три раза с разными нормалями). Пропускаем создание буфера вершин и связывание его с графическим конвейером, там ничего не меняется, и сейчас нам это не интересно. Нам интересно, как создается индексный буфер. Начинается все так же, как

при создании вершинного буфера, с заполнения массива. В этот массив мы должны поместить индексы вершин, указывающие на элементы из вершинного буфера. Здесь нужно иметь в виду, что отсчет элементов начинается от нуля, то есть первый элемент буфера вершин имеет индекс 0. Еще один важный момент: порядок, в котором мы перечисляем вершины треугольников. Вершины должны указываться по часовой стрелке, если смотреть на треугольник с видимой стороны (порядок вершин используется при отсечении невидимыхых граней, помните?). Давайте разберемся, какие индексы нужны, чтобы построить верхнюю грань куба (рис. 7.11).

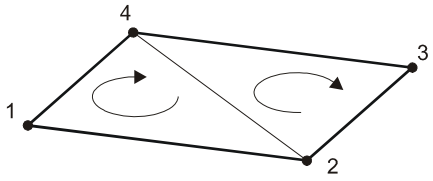

Рис. 7.11. Треугольники, образующие верхнюю грань куба

Для начала запишем, какие вершины образуют левый и правый треугольники. Например, таким образом: левый треугольник образован вершинами 4, 2, 1, правый — вершинами 3, 2, 4. Можно начать обход и с другой вершины, лишь бы его направление было по часовой стрелке. Номера вершин у нас есть, чтобы получить индексы, по очереди запишем индексы соответствующих элементов массива `vertices[]`:

```
3, 1, 0,
2, 1, 3.
```

Так как индексы представляют собой целые числа, объявим массив из элементов типа `DWORD` и занесем туда индексы вершин для всех двенадцати треугольников, образующих куб. Массив со всеми индексами выглядит так:

```
DWORD indices[] =
{
3, 1, 0,
2, 1, 3,

6, 4, 5,
7, 4, 6,

11, 9, 8,
10, 9, 11,
```

```
14, 12, 13,
15, 12, 14,

19, 17, 16,
18, 17, 19,

22, 20, 21,
23, 20, 22
};
```

Для создания индексного буфера необходимо заполнить структуру `D3D10_BUFFER_DESC`, аналогично тому, как мы делали это для буфера вершин. Конечно, для индексного буфера некоторые поля будут отличаться. Первое поле остается без изменений:

```
bd.Usage = D3D10_USAGE_DEFAULT;
```

В следующем поле мы указываем количество памяти в байтах, которое необходимо выделить для индексного буфера:

```
bd.ByteWidth = sizeof(DWORD)*36;
```

Далее определяем, в каком качестве создаваемый буфер будет связан с графическим конвейером, для чего указываем флаг, означающий "индексный буфер":

```
bd.BindFlags = D3D10_BIND_INDEX_BUFFER;
```

Два поля оставим без изменений:

```
bd.CPUAccessFlags = 0;
bd.MiscFlags = 0;
```

Затем впишем указатель на данные, которыми будет заполнен индексный буфер при инициализации, то есть на массив `indices[]`:

```
InitData.pSysMem = indices;
```

Теперь мы можем создать индексный буфер и связать его с графическим конвейером. Ничего нового тут нет, буфер создается так:

```
hr = g_pd3dDevice->CreateBuffer( &bd, &InitData, &g_pIndexBuffer );
if( FAILED(hr) )
  return hr;
```

А вот чтобы связать индексный буфер с графическим конвейером, необходимо вызвать метод `ID3D10Device::IASetIndexBuffer`, и он несколько отличается по входным параметрам от аналогичного метода для буфера вершин:

```
void IASetIndexBuffer(
  ID3D10Buffer *pIndexBuffer,
```

```
  DXGI_FORMAT Format,
  UINT Offset
);
```

Параметры требуется указать следующие:

- `pIndexBuffer` — указатель на буфер, содержащий индексы вершин;
- `Format` — формат данных индексного буфера, допустимыми в данном случае являются только два значения: `DXGI_FORMAT_R16_UINT` и `DXGI_FORMAT_R32_UINT`;
- `Offset` — смещение в байтах от начала индексного буфера до первой используемой вершины.

Свяжем индексный буфер с графическим конвейером:

```
g_pd3dDevice->
        IASetIndexBuffer( g_pIndexBuffer, DXGI_FORMAT_R32_UINT, 0 );
```

В оставшихся строках функции изменений не будет, разве что немного подкорректируем положение камеры, чтобы наблюдать куб с более высокой точки:

```
D3DXVECTOR3 Eye( 0.0f, 2.0f, -5.0f );
```

Чтобы более наглядно представить себе изменения в функции, приведем ее новую версию целиком:

```
HRESULT InitDirect3D10()
{
  HRESULT hr = S_OK;

  // Размеры клиентской области окна
  RECT rc;
  GetClientRect( g_hWnd, &rc );
  UINT width = rc.right - rc.left;
  UINT height = rc.bottom - rc.top;

  // Список возможных типов устройства
  D3D10_DRIVER_TYPE driverTypes[] =
  {
    D3D10_DRIVER_TYPE_HARDWARE,
    D3D10_DRIVER_TYPE_REFERENCE,
  };
  UINT numDriverTypes =sizeof(driverTypes) / sizeof(driverTypes[0]);

  // Заполняем структуру
  DXGI_SWAP_CHAIN_DESC sd;
```

```
ZeroMemory( &sd, sizeof(sd) );
sd.BufferCount = 1;
sd.BufferDesc.Width = width;
sd.BufferDesc.Height = height;
sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
sd.BufferDesc.RefreshRate.Numerator = 60;
sd.BufferDesc.RefreshRate.Denominator = 1;
sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
sd.OutputWindow = g_hWnd;
sd.SampleDesc.Count = 1;
sd.SampleDesc.Quality = 0;
sd.Windowed = TRUE;

// Пытаемся создать устройство, проходя по списку
// Как только получилось - выходим из цикла
for( UINT driverTypeIndex = 0; driverTypeIndex < numDriverTypes;
            driverTypeIndex++ )
{
  g_driverType = driverTypes[driverTypeIndex];
  hr = D3D10CreateDeviceAndSwapChain( NULL, g_driverType, NULL, 0,
    D3D10_SDK_VERSION, &sd, &g_pSwapChain, &g_pd3dDevice );
  if( SUCCEEDED( hr ) )
    break;
}
if( FAILED(hr) )
  return hr;

// Представление данных для
// буфера визуализации
ID3D10Texture2D *pBackBuffer;
hr = g_pSwapChain->GetBuffer( 0, __uuidof( ID3D10Texture2D ),
                              (LPVOID*)&pBackBuffer );
if( FAILED(hr) )
  return hr;
// Создадим представление данных
hr = g_pd3dDevice->CreateRenderTargetView( pBackBuffer, NULL,
                                           &g_pRenderTargetView );
pBackBuffer->Release();
if( FAILED(hr) )
```

```
    return hr;

  // Свяжем буфер визуализации с графическим конвейером
  g_pd3dDevice->OMSetRenderTargets( 1, &g_pRenderTargetView, NULL );

  // Настроим область отображения
  D3D10_VIEWPORT vp;
  vp.Width = width;
  vp.Height = height;
  vp.MinDepth = 0.0f;
  vp.MaxDepth = 1.0f;
  vp.TopLeftX = 0;
  vp.TopLeftY = 0;
  g_pd3dDevice->RSSetViewports( 1, &vp );

  // Создадим эффект
  hr = D3DX10CreateEffectFromFile( L"DirectLight.fx", NULL, NULL,
                    "fx_4_0", D3D10_SHADER_ENABLE_STRICTNESS, 0,
                    g_pd3dDevice, NULL, NULL, &g_pEffect, NULL, NULL );

  if( FAILED( hr ) )
  {
    MessageBox( NULL, L"Не удается обнаружить файл эффектов (FX).
Файл эффектов должен находиться в той же папке, что и исполняемый
файл", L"Ошибка", MB_OK );
    return hr;
  }

  // Извлекаем технику отображения
  g_pTechnique = g_pEffect->GetTechniqueByName( "RenderDirectLight" );

  // Связь с переменными шейдера
  g_pWorldVariable = g_pEffect->GetVariableByName( "World" )->AsMatrix();
  g_pViewVariable = g_pEffect->GetVariableByName( "View" )->AsMatrix();
  g_pProjectionVariable = g_pEffect->
                      GetVariableByName( "Projection" )->AsMatrix();
  g_pLightColor = g_pEffect->
                      GetVariableByName( "LightColor" )->AsVector();
  g_pLightDirection = g_pEffect->
```

```
                        GetVariableByName( "LightDirection" )->AsVector();

  // Описываем формат входных данных
  D3D10_INPUT_ELEMENT_DESC layout[] =
  {
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
      D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
      D3D10_INPUT_PER_VERTEX_DATA, 0 }
  };
  UINT numElements = sizeof(layout)/sizeof(layout[0]);

  // Создаем объект входных данных
  D3D10_PASS_DESC PassDesc;
  g_pTechnique->GetPassByIndex( 0 )->GetDesc( &PassDesc );
  hr = g_pd3dDevice->CreateInputLayout( layout, numElements,
          PassDesc.pIAInputSignature, PassDesc.IAInputSignatureSize,
          &g_pVertexLayout );
  if( FAILED( hr ) )
    return hr;

  // Связываем объект входных данных с графическим конвейером
  g_pd3dDevice->IASetInputLayout( g_pVertexLayout );

  // Создаем буфер вершин
SimpleVertex vertices[] =
{
{D3DXVECTOR3(-1.0f, 1.0f,-1.0f ), D3DXVECTOR3( 0.0f, 1.0f, 0.0f )},
{D3DXVECTOR3( 1.0f, 1.0f,-1.0f ), D3DXVECTOR3( 0.0f, 1.0f, 0.0f )},
{D3DXVECTOR3( 1.0f, 1.0f, 1.0f ), D3DXVECTOR3( 0.0f, 1.0f, 0.0f )},
{D3DXVECTOR3(-1.0f, 1.0f, 1.0f ), D3DXVECTOR3( 0.0f, 1.0f, 0.0f )},

{D3DXVECTOR3(-1.0f,-1.0f,-1.0f ), D3DXVECTOR3( 0.0f, -1.0f, 0.0f )},
{D3DXVECTOR3( 1.0f,-1.0f,-1.0f ), D3DXVECTOR3( 0.0f, -1.0f, 0.0f )},
{D3DXVECTOR3( 1.0f,-1.0f, 1.0f ), D3DXVECTOR3( 0.0f, -1.0f, 0.0f )},
{D3DXVECTOR3(-1.0f,-1.0f, 1.0f ), D3DXVECTOR3( 0.0f, -1.0f, 0.0f )},

{D3DXVECTOR3(-1.0f,-1.0f, 1.0f ), D3DXVECTOR3(-1.0f, 0.0f, 0.0f )},
{D3DXVECTOR3(-1.0f,-1.0f,-1.0f ), D3DXVECTOR3(-1.0f, 0.0f, 0.0f )},
```

```
{D3DXVECTOR3(-1.0f, 1.0f,-1.0f ), D3DXVECTOR3(-1.0f, 0.0f, 0.0f )},
{D3DXVECTOR3(-1.0f, 1.0f, 1.0f ), D3DXVECTOR3(-1.0f, 0.0f, 0.0f )},

{D3DXVECTOR3( 1.0f,-1.0f, 1.0f ), D3DXVECTOR3( 1.0f, 0.0f, 0.0f )},
{D3DXVECTOR3( 1.0f,-1.0f,-1.0f ), D3DXVECTOR3( 1.0f, 0.0f, 0.0f )},
{D3DXVECTOR3( 1.0f, 1.0f,-1.0f ), D3DXVECTOR3( 1.0f, 0.0f, 0.0f )},
{D3DXVECTOR3( 1.0f, 1.0f, 1.0f ), D3DXVECTOR3( 1.0f, 0.0f, 0.0f )},

{D3DXVECTOR3(-1.0f,-1.0f,-1.0f ), D3DXVECTOR3( 0.0f, 0.0f,-1.0f )},
{D3DXVECTOR3( 1.0f,-1.0f,-1.0f ), D3DXVECTOR3( 0.0f, 0.0f,-1.0f )},
{D3DXVECTOR3( 1.0f, 1.0f,-1.0f ), D3DXVECTOR3( 0.0f, 0.0f,-1.0f )},
{D3DXVECTOR3(-1.0f, 1.0f,-1.0f ), D3DXVECTOR3( 0.0f, 0.0f,-1.0f )},

{D3DXVECTOR3(-1.0f,-1.0f, 1.0f ), D3DXVECTOR3( 0.0f, 0.0f, 1.0f )},
{D3DXVECTOR3( 1.0f,-1.0f, 1.0f ), D3DXVECTOR3( 0.0f, 0.0f, 1.0f )},
{D3DXVECTOR3( 1.0f, 1.0f, 1.0f ), D3DXVECTOR3( 0.0f, 0.0f, 1.0f )},
{D3DXVECTOR3(-1.0f, 1.0f, 1.0f ), D3DXVECTOR3( 0.0f, 0.0f, 1.0f )},
};

  D3D10_BUFFER_DESC bd;
  bd.Usage = D3D10_USAGE_DEFAULT;
  bd.ByteWidth = sizeof( SimpleVertex ) * 24;
  bd.BindFlags = D3D10_BIND_VERTEX_BUFFER;
  bd.CPUAccessFlags = 0;
  bd.MiscFlags = 0;
  D3D10_SUBRESOURCE_DATA InitData;
  InitData.pSysMem = vertices;
  hr = g_pd3dDevice->CreateBuffer( &bd, &InitData, &g_pVertexBuffer );
  if( FAILED( hr ) )
    return hr;

  // Связываем буфер вершин с графическим конвейером
  UINT stride = sizeof( SimpleVertex );
  UINT offset = 0;
  g_pd3dDevice->IASetVertexBuffers( 0, 1, &g_pVertexBuffer,
                                    &stride, &offset );

  // Создаем индексный буфер
  DWORD indices[] =
```

```
{
  3, 1, 0,
  2, 1, 3,

  6, 4, 5,
  7, 4, 6,

  11, 9, 8,
  10, 9, 11,

  14, 12, 13,
  15, 12, 14,

  19, 17, 16,
  18, 17, 19,

  22, 20, 21,
  23, 20, 22
};
bd.Usage = D3D10_USAGE_DEFAULT;
bd.ByteWidth = sizeof( DWORD ) * 36;
// (нужно 36 вершин для 12 треугольников)
bd.BindFlags = D3D10_BIND_INDEX_BUFFER;
bd.CPUAccessFlags = 0;
bd.MiscFlags = 0;
InitData.pSysMem = indices;
hr = g_pd3dDevice->CreateBuffer( &bd, &InitData, &g_pIndexBuffer );
if( FAILED(hr) )
  return hr;

// Свяжем индексный буфер с конвейером
g_pd3dDevice->
       IASetIndexBuffer( g_pIndexBuffer, DXGI_FORMAT_R32_UINT, 0 );

// Задаем тип и способ построения примитивов
g_pd3dDevice->
       IASetPrimitiveTopology( D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST );

// Инициализируем мировую матрицу
D3DXMatrixIdentity( &g_World );
```

```
  // Инициализируем матрицу вида
  D3DXVECTOR3 Eye( 0.0f, 2.0f, -5.0f );
  D3DXVECTOR3 At( 0.0f, 0.0f, 0.0f );
  D3DXVECTOR3 Up( 0.0f, 1.0f, 0.0f );
  D3DXMatrixLookAtLH( &g_View, &Eye, &At, &Up );

  // Инициализируем матрицу проекции
  D3DXMatrixPerspectiveFovLH( &g_Projection, (float)D3DX_PI * 0.25f,
                              width/(FLOAT)height, 0.1f, 100.0f );

  return S_OK;
}
```

Самую долгую и монотонную часть работы мы закончили. По сравнению с ней редактирование функции `RenderScene()` — настоящий отдых. В ней нам предстоит лишь организовать передачу в шейдер данных об источнике света и поменять метод рисования на индексный. Откроем функцию `RenderScene()` и найдем ту часть, где производится передача значений в переменные шейдера. Сначала зададим цвет света, который излучается источником, у нас это будет серый цвет, для более приятного визуального результата:

```
float LightColor[4]={0.5f, 0.5f, 0.5f, 1.0f};
```

Передадим информацию о цвете в шейдер:

```
g_pLightColor->SetFloatVector((float*)&LightColor);
```

Аналогичным образом поступим с положением источника света в пространстве:

```
float LightDir[4] = {0.0f, 0.0f, -1.5f, 1.0f};
g_pLightDirection->SetFloatVector((float*)&LightDir);
```

Изменим метод, которым производится рисование куба, применив метод `ID3D10Device::DrawIndexed`. Он немного отличается от метода, который мы использовали раньше. Вот прототип этого метода:

```
void DrawIndexed(
  UINT IndexCount,
  UINT StartIndexLocation,
  INT BaseVertexLocation
);
```

Опишем параметры, которые передаются методу:

- `IndexCount` — количество индексов вершин для прорисовки;
- `StartIndexLocation` — индекс первого индекса вершины, с которого начинается прорисовка (полезно в том случае, если мы храним в одном индексном буфере описание нескольких объектов);

- `BaseVertexLocation` — индекс первой вершины в вершинном буфере, который добавляется к каждому индексу вершины, чтобы получить соответствующий элемент из вершинного буфера. Он может принимать отрицательные значения, что тоже может быть полезным при хранении в вершинном буфере нескольких объектов.

Для того чтобы нарисовать куб, необходимо передать информацию о 36 вершинах, которые образуют грани куба. Таким образом, первый параметр нам известен. Остальные параметры установим равными нулю, так как используем данные в буферах без сдвигов. Вызов метода будет выглядеть следующим образом:

```
g_pd3dDevice->DrawIndexed( 36, 0, 0 );
```

После исправления функция должна выглядеть так:

```
void RenderScene()
{
  // Счетчик времени
  static float t = 0.0f;
  if( g_driverType == D3D10_DRIVER_TYPE_REFERENCE )
  {
    t += (float)D3DX_PI * 0.0125f;
  }
  else
  {
    static DWORD dwTimeStart = 0;
    DWORD dwTimeCur = GetTickCount();
    if( dwTimeStart == 0 )
      dwTimeStart = dwTimeCur;
    t = ( dwTimeCur - dwTimeStart) / 1000.0f;
  }

  // Матрица поворота куба
  // на угол в зависимости от времени t
  D3DXMatrixRotationY( &g_World, t );

  // Очищаем вторичный буфер
  //(компоненты красного, зеленого, синего, прозрачность)
  float ClearColor[4] = { 0.1f, 0.5f, 0.1f, 1.0f };
  g_pd3dDevice->
         ClearRenderTargetView( g_pRenderTargetView, ClearColor );
```

```
  // Передаем информацию о матрицах преобразования
  g_pWorldVariable->SetMatrix( (float*)&g_World );
  g_pViewVariable->SetMatrix( (float*)&g_View );
  g_pProjectionVariable->SetMatrix( (float*)&g_Projection );

  // Передаем информацию об источнике света
  float LightColor[4]={0.5f, 0.5f, 0.5f, 1.0f};
  g_pLightColor->SetFloatVector((float*)&LightColor);

  float LightDir[4] = {0.0f, 0.0f, -1.5f, 1.0f};
  g_pLightDirection->SetFloatVector((float*)&LightDir);

  // Рисуем куб
  D3D10_TECHNIQUE_DESC techDesc;
  g_pTechnique->GetDesc( &techDesc );
  for( UINT p = 0; p < techDesc.Passes; ++p )
  {
    g_pTechnique->GetPassByIndex( p )->Apply(0);
    g_pd3dDevice->DrawIndexed( 36, 0, 0 );
  }
  // Переключаем буферы
  g_pSwapChain->Present( 0, 0 );
}
```

На этом все, подготовка приложения закончена. Теперь нужно написать шейдеры, которые реализуют выбранную нами модель освещения и с помощью которых мы увидим, как наш виртуальный источник света виртуально освещает виртуальный куб.

Направленный источник света

Приступаем к реализации модели освещения. Как мы уже не раз имели возможность убедиться, вся связанная с графикой работа происходит в шейдерах. Что у нас изменилось по сравнению с шейдерами, которые мы писали до этого? Сейчас у нас в шейдер передаются координаты нормалей. Запишем структуру входных параметров вершинного шейдера с учетом этого нововведения:

```
struct VS_INPUT
{
    float4 Pos : POSITION;
    float3 Normal : NORMAL;
};
```

Думаю, в комментариях эта структура не нуждается. Аналогичным образом записывается структура для входных параметров пиксельного шейдера:

```
struct PS_INPUT
{
    float4 Pos : SV_POSITION;
    float3 Normal : TEXTCOORD0;
};
```

Вот тут комментарии потребуются. Зачем передавать координаты нормалей в пиксельный шейдер? Правильно, для того, чтобы рассчитать освещенность точки согласно уравнению нашей модели освещения. А почему с нормалями связано семантическое имя `TEXTCOODR0`? Дело в том, что пиксельный шейдер не может иметь у входных параметров семантику `NORMAL`, поэтому мы и передаем эти данные под видом текстурных координат.

Не забудем про глобальные переменные. Те из них, что присутствовали в предыдущих шейдерах, нам, конечно же, понадобятся и здесь:

```
matrix World;
matrix View;
matrix Projection;
```

К ним мы добавим две новые переменные, через которые передаются данные об источнике света:

```
float4 LightColor;
float4 LightDirection;
```

Функция вершинного шейдера не сильно отличается от своих предыдущих версий. Заголовок функции остается прежним, без изменений остается и умножение координат вершины в пространстве на матрицы преобразований:

```
PS_INPUT VS( VS_INPUT Data )
{
    PS_INPUT Out;
    Out = (PS_INPUT)0;

     // Умножаем координаты вершины
    // на матрицы преобразований
    Out.Pos = mul( Data.Pos, World );
    Out.Pos = mul( Out.Pos, View );
    Out.Pos = mul( Out.Pos, Projection );
```

Координаты нормалей также нужно преобразовать, но умножить их нужно только на мировую матрицу.

```
Out.Normal = mul( Data.Normal, World );
```

Вернем результат работы функции:

```
    return Out;
}
```

В функции пиксельного шейдера будет производиться расчет освещения для каждого пиксела индивидуально. При этом будут использоваться переданные из вершинного шейдера координаты нормалей, преобразованные в пространство мировых координат. Посмотрим сразу на всю функцию целиком:

```
float4 PS_DirectLight( PS_INPUT Vertex ) : SV_Target
{
    float4 FinalColor =
saturate( LightColor * dot( (float3)LightDirection, Vertex.Normal ) );
    FinalColor.a = 1;

    return FinalColor;
}
```

Расчет цвета пиксела производится в самой первой строке функции. Как он происходит? Сначала с помощью функции `dot()` вычисляется скалярное произведение векторов, задающих направление лучей от источника света и направление нормали к поверхности. Нормаль интерполируется для каждого пиксела треугольника, исходя из направления нормалей у трех его вершин. В нашем случае все нормали будут направлены в одну сторону. Так как модули векторов равны единице, результат произведения будет равен значению косинуса угла между ними. Скалярное произведение, согласно закону косинусов Ламберта, представляет собой коэффициент, показывающий количество отраженного от поверхности света. На этот коэффициент мы умножаем цвет света от направленного источника и получаем цвет рассчитываемого пиксела. Составляющие полученного цвета пиксела с помощью функции `saturate()` приводятся к интервалу от 0 до 1. Кто-нибудь уже наверняка заметил, что согласно выбранной модели освещения в произведении должен также присутствовать цвет материала самого куба. Для простоты изложения мы считаем цвет куба белым, то есть цвет куба у нас учитывается, просто все его составляющие равны 1.0, и мы просто опускаем умножение на единицу. В следующей строке мы явно устанавливаем непрозрачность для цвета пиксела. При необходимости, она также может быть результатом вычислений.

Шейдеры мы написали, и осталось только описать технику отображения. Здесь все аналогично предыдущим файлам эффектов, мы лишь изменим имя техники отображения на "RenderDirectLight":

```
technique10 RenderDirectLight
{
```

```
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, VS() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, PS_DirectLight() ) );
    }
}
```

После этого нам остается только сохранить исходный текст файла эффектов в файл "DirectLight.fx" и откомпилировать и запустить проект. Готовый проект и файл эффектов находятся на компакт-диске в директории Glava7\DirectLight. Если мы все сделали правильно, то мы увидим вращающийся куб, освещенность граней которого меняется в зависимости от угла поворота (рис. 7.12).

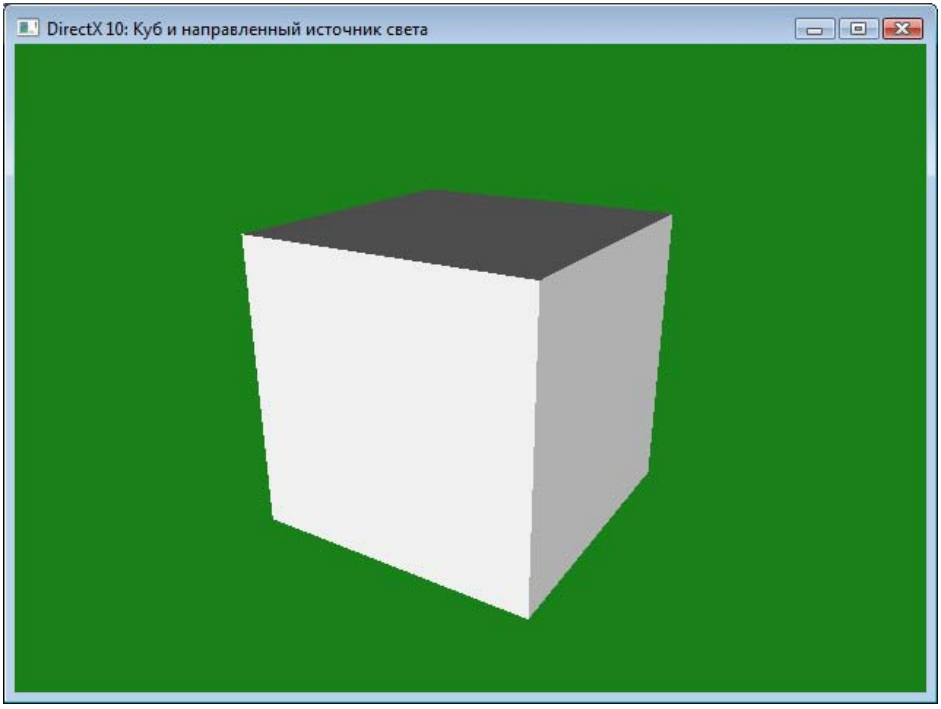

Рис. 7.12. Результат работы шейдера для направленного источника света.

Конечно, каждая грань окрашена в один однородный цвет, что не придает желаемого реализма. Но ведь это только наш первый шаг при работе с освещением. Давайте двигаться дальше, мы еще сможем улучшить очень многое.

Точечный источник света

Точечный источник света должен придать нашей сцене более реалистичный вид, так как он дает неравномерную освещенность, которая по эту сторону экрана встречается гораздо чаще. Что требуется сделать, чтобы воплотить нашу идею? Правильно, для начала создаем новый проект и копируем в него исходный текст из проекта с направленным источником света, также перепишем оттуда файл "DirectLight.fx" и переименуем его в "PointLight.fx". Еще нам нужно добавить в программу переменную для хранения координат положения точечного источника света в пространстве. Можно просто переименовать переменную, в которой мы до этого хранили направление лучей, для точечного источника она нам не понадобится. Как бы то ни было, в объявлении глобальных переменных для нового источника света должны присутствовать две строки:

```
ID3D10EffectVectorVariable* g_pLightColor = NULL;
ID3D10EffectVectorVariable* g_pLightPosition = NULL;
```

После глобальных переменных традиционно обращаемся к функции `InitDirect3D10()`, на этот раз работы здесь не много. Исправляем имя файла эффектов в параметрах функции `D3DX10CreateEffectFromFile()` на новое (да, это "PointLight.fx") и меняем название техники отображения на "RenderPointLight". После этого устанавливаем связь с переменными шейдера, для переменных источника света мы опять же добавляем строки:

```
g_pLightColor = g_pEffect->GetVariableByName( "LightColor" )->AsVector();
g_pLightPosition = g_pEffect->
                        GetVariableByName( "LightPosition" )->AsVector();
```

Остальная часть функции исправлений не требует, и мы перемещаемся в функцию `RenderScene()`. В ней мы должны передать значения переменным, связь с которыми мы только что установили. Выглядеть это будет следующим образом. Сначала передаем данные о цвете светового источника (этот фрагмент — такой же, как и для направленного источника света):

```
float LightColor[4]={1.0f, 1.0f, 1.0f, 1.0f};
g_pLightColor->SetFloatVector((float*)&LightColor);
```

Следом передаем координаты источника в пространстве:

```
float LightPos[4] = {1.0f, 1.0f, -2.0f, 1.0f};
g_pLightPosition->SetFloatVector((float*)&LightPos);
```

На этом изменения текста основной программы закончены. Переходим к самому интересному — к составлению шейдера. Сразу подкорректируем глобальные переменные, чтобы потом к этому не возвращаться: исправим имя

переменной с "LightDirection" на "LightPosition". Вот как теперь должно выглядеть объявление глобальных переменных в файле эффектов:

```
matrix World;
matrix View;
matrix Projection;

float4 LightColor;
float4 LightPosition;
```

Сейчас, когда имена переменных приведены в соответствие с объявленными в тексте основной программы, давайте представим, как должен работать наш новый шейдер. Напомню основное уравнение нашей модели освещения: $I_o = D \cdot I_u \cdot \mathrm{Cos}(\theta)$. В рассматриваемом нами случае дело еще осложняется тем, что интенсивность света от источника зависит от расстояния до него. То есть уравнение освещенности точки принимает следующий вид:

$$I_o = D \cdot \frac{I_u}{d^2} \cdot \mathrm{Cos}(\theta),$$

где d — расстояние от точки до источника света.

Основным отличием от шейдера с направленным источником света будет отсутствие одного общего направления для лучей света. Вектор направления на источник придется вычислять для каждой точки индивидуально. Для этого придется передавать в пиксельный шейдер координаты вершины, умноженные на мировую матрицу. При передаче все данные интерполируются для текущего рассчитываемого пиксела, то есть у нас получатся мировые координаты точки на грани куба, куда приходит луч света. Ну, а уж найти координаты вектора, зная координаты его начала и конца, для нас труда не составит. Скалярное произведение этого вектора на нормаль даст нам количество отраженного света в данной точке. Модуль этого вектора представляет собой расстояние от точки до источника света. Это очень полезная информация, особенно если учесть, что в зависимости от этого расстояния меняется интенсивность света.

Принимая в расчет все изложенные соображения, начнем создавать наш шейдер. Начнем со структур входных параметров. Входные параметры для вершинного шейдера остались прежними, эту структуру мы не трогаем. А вот в структуру входных данных пиксельного шейдера мы должны добавить еще одно поле, назовем его `InterPos`. Оно будет использоваться для передачи координат вершины в мировом пространстве в пиксельный шейдер. Обновленная структура будет выглядеть так (в качестве семантического

имени мы снова используем имя для текстурных координат, но с другим индексом):

```
struct PS_INPUT
{
    float4 Pos : SV_POSITION;
    float3 InterPos : TEXTCOORD0;
    float3 Normal : TEXTCOORD1;
};
```

Новый текст вершинного шейдера выглядит следующим образом:

```
PS_INPUT VS( VS_INPUT Data )
{
  PS_INPUT Out;
  Out = (PS_INPUT)0;

  Out.Pos = mul (Data.Pos, World);

  Out.InterPos = Out.Pos.xyz;

  Out.Pos = mul (Out.Pos, View);
  Out.Pos = mul (Out.Pos, Projection);

  Out.Normal = mul (Data.Normal, World);

  return Out;
}
```

Вот что делает этот шейдер. Сперва он умножает переданные на вход координаты вершины на мировую матрицу и результат сохраняет в структуре выходных параметров в поле `InterPos`. Далее координаты вершины умножаются на матрицу вида и матрицу проекции, и результат сохраняется в поле `Pos`. В завершение функция производит умножение координат нормали на мировую матрицу и возвращает результат проделанных вычислений. Вершинный шейдер несложный, и совсем немного отличается от предыдущего варианта.

Другое дело — пиксельный шейдер. Давайте пошагово рассмотрим, как происходит вычисление цвета пиксела. В первую очередь мы вычисляем координаты вектора, направленного из текущей точки в точку нахождения источника света:

```
float3 PixelToLight = (float3)LightPosition - Vertex.InterPos;
```

После этого производим нормализацию интерполированного вектора нормали, так как при интерполяции длина вектора может исказиться:

```
float3 NewNormal = normalize(Vertex.Normal);
```

Затем с помощью нормализации получаем единичный вектор, направленный на источник света, который потребуется при вычислении скалярного произведения:

```
float3 NewDirection = normalize(PixelToLight);
```

После этого вычисляем интенсивность на том расстоянии, на которое текущая точка удалена от источника света (функция `length()` возвращает модуль вектора):

```
float4 LightIntensity =LightColor/pow(length(PixelToLight),2.0);
```

Наконец, получаем цвет пиксела, используя следующее выражение:

```
float4 FinalColor = saturate( LightIntensity*
                              dot( NewNormal, NewDirection) );
```

Все то же самое, что и для направленного источника света, только расчет составляющих выражения немного усложнился. Для полученного цвета устанавливаем полную непрозрачность, как мы делали в предыдущем примере:

```
FinalColor.a = 1;
```

Все, цвет пиксела вычислен, и мы возвращаем результат работы функции:

```
return FinalColor;
```

Полный текст файла эффектов приведен в листинге 7.1. Соответствующий файл эффектов и проект находятся на компакт-диске в папке \Glava7\ PointLight.

Листинг 7.1

```
// Шейдеры для точечного источника света
// Файл PointLight.fx
//--------------------------------------------------------------------------
// Структуры данных
//--------------------------------------------------------------------------
struct VS_INPUT
{
    float4 Pos : POSITION;
    float3 Normal : NORMAL;
};

struct PS_INPUT
```

```
{
    float4 Pos : SV_POSITION;
    float3 InterPos : TEXTCOORD0;
    float3 Normal : TEXTCOORD1;
};
//--------------------------------------------------------------------------
// Глобальные переменные
//--------------------------------------------------------------------------
matrix World;
matrix View;
matrix Projection;

float4 LightColor;
float4 LightPosition;

//--------------------------------------------------------------------------
// Функция вершинного шейдера
//--------------------------------------------------------------------------
PS_INPUT VS( VS_INPUT Data )
{
    PS_INPUT Out;
    Out = (PS_INPUT)0;

    // Умножим координаты вершины
    // на матрицу преобразований
    Out.Pos = mul (Data.Pos, World);

    // Подготовим эти координаты для
    // передачи в пиксельный шейдер
    Out.InterPos = Out.Pos.xyz;

    // Умножим координаты на
    // оставшиеся матрицы
    Out.Pos = mul (Out.Pos, View);
    Out.Pos = mul (Out.Pos, Projection);

    Out.Normal = mul (Data.Normal, World);

    return Out;
}
```

```
//--------------------------------------------------------------------------
// Функция пиксельного шейдера
//--------------------------------------------------------------------------
float4 PS_PointLight( PS_INPUT Vertex ) : SV_Target
{
    // Вектор из точки к источнику света
    float3 PixelToLight = (float3)LightPosition - Vertex.InterPos;

    // Нормализуем интерполированный вектор
    float3 NewNormal = normalize(Vertex.Normal);
    // Единичный вектор, направленный на источник света
    float3 NewDirection = normalize(PixelToLight);
    // Интенсивность света с учетом расстояния
    float4 LightIntensity =LightColor/pow(length(PixelToLight),2.0);

    // Итоговый цвет пиксела
    float4 FinalColor = saturate( LightIntensity*
                                  dot(NewNormal, NewDirection) );
    FinalColor.a = 1;

    return FinalColor;
}
//--------------------------------------------------------------------------
// Техника отображения
//--------------------------------------------------------------------------
technique10 RenderPointLight
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, VS() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, PS_PointLight() ) );
    }
}
```

Откомпилируем и запустим проект. В результате работы программы у нас должно получиться изображение вращающегося куба с реалистичным освещением граней (рис. 7.13). Поэкспериментируйте, задавая различные координаты расположения источника света и составляющих цвета, чтобы получить

наилучший результат. А у нас остался последний нерассмотренный источник света — прожектор. Им мы сейчас и займемся.

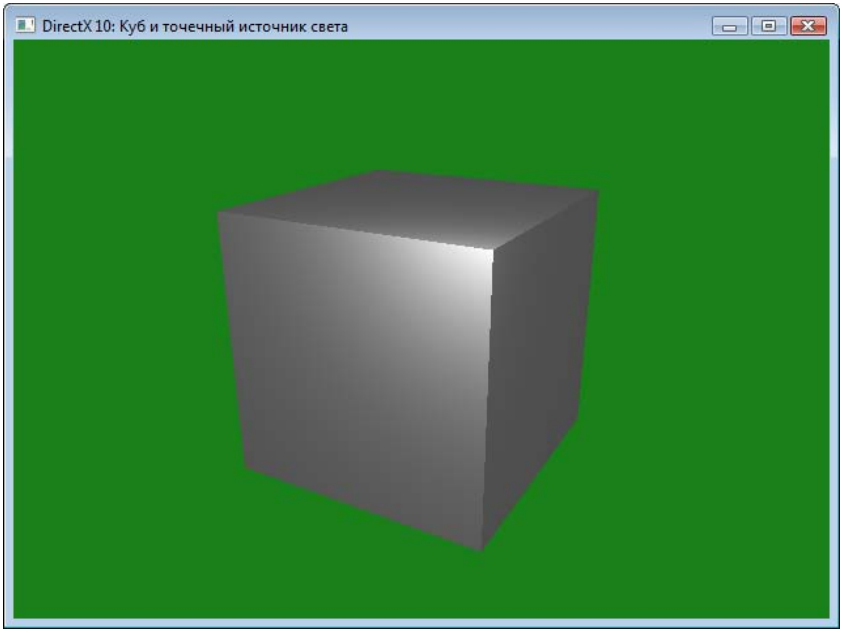

Рис. 7.13. Реалистичное освещение куба точечным источником

Прожектор

Из рассматриваемых нами источников света прожектор является самым сложным для моделирования. Освежим в памяти рис. 7.4, он нам сейчас очень пригодится. В двух словах повторим свойства источника света. Интенсивность его света зависит от расстояния до точки и от того, в каком из световых конусов она находится (от угла α). Запишем уравнение расчета освещенности точки для прожектора в упрощенном виде:

$$I_o = I_y \cdot I_p \cdot \mathrm{Cos}(\theta),$$

где

- I_y — интенсивность света в зависимости от светового конуса, в котором находится точка (угла α);
- I_p — интенсивность света в зависимости от удаленности точки от источника света.

При такой записи все выглядит довольно просто, но при реализации будет немного сложнее. Начнем мы, как обычно, с подготовки исходного текста основной программы. Создадим новый проект, назовем его "SpotLight" и скопируем в него исходный текст из предыдущего проекта. В папку, где расположен проект, скопируем файл эффектов "PointLight.fx" и переименуем его в "SpotLight.fx". Вы уже, наверное, и сами знаете, что делать дальше. Для источника света типа прожектор нам понадобятся новые переменные. Надо будет добавить переменную для хранения вектора направления света и для углов световых конусов. Открываем объявление глобальных переменных программы и добавляем их. Для описания источника света должны быть объявлены следующие переменные:

```
ID3D10EffectVectorVariable* g_pLightColor = NULL;
ID3D10EffectVectorVariable* g_pLightPosition = NULL;
ID3D10EffectVectorVariable* g_pLightDirection = NULL;
ID3D10EffectScalarVariable* g_pSpotLightPhi = NULL;
ID3D10EffectScalarVariable* g_pSpotLightTheta = NULL;
```

Переходим в функцию `InitDirect3D10()`, корректируем, какой файл эффектов загружать ("SpotLight.fx") и какую технику отображения из него извлекать ("RenderSpotLight"). Устанавливаем связь с переменными шейдера, для всех переменных источника света это выглядит следующим образом:

```
g_pLightColor = g_pEffect->GetVariableByName( "LightColor" )->AsVector();
g_pLightPosition = g_pEffect->
                        GetVariableByName( "LightPosition" )->AsVector();
g_pLightDirection = g_pEffect->
                        GetVariableByName( "LightDirection" )->AsVector();
g_pSpotLightPhi = g_pEffect->GetVariableByName( "Phi" )->AsScalar();
g_pSpotLightTheta = g_pEffect->GetVariableByName( "Theta" )->AsScalar();
```

Далее отправляемся в функцию `RenderScene()`, где устанавливаем значения этих и других переменных и передаем их на обработку шейдерам. Вот как мы это сделаем для переменных — параметров источника света:

```
float LightColor[4]={1.0f, 0.3f, 0.3f, 1.0f};
g_pLightColor->SetFloatVector((float*)&LightColor);

float LightPos[4] = {0.0f, 0.0f, -1.5f, 1.0f};
g_pLightPosition->SetFloatVector((float*)&LightPos);

float LightDirection[4] = {0.0f,0.0f,10.0f,1.0f};
g_pLightDirection->SetFloatVector((float*)&LightDirection);
```

Значения углов для внутреннего и наружного световых конусов преобразуем в радианы, так как именно в таком виде требуется указывать параметры для всех тригонометрических функций HLSL. Для преобразования воспользуемся стандартным макросом `D3DXToRadian()`:

```
float Phi = D3DXToRadian(50.0);// угол в радианах
float Theta = D3DXToRadian(25.0);// угол в радианах
g_pSpotLightPhi->SetFloat(Phi);
g_pSpotLightTheta->SetFloat(Theta);
```

Основная программа готова. Осталось написать соответствующие шейдеры, где реализуется алгоритм расчета освещенности. Откроем файл "SpotLight.fx" и исправим его под новый тип источника света. Структуры входных параметров останутся без изменения. В объявление глобальных переменных добавляем переменные для хранения характеристик прожектора. Это направление, в котором нацелен прожектор и углы световых конусов. Полностью все глобальные переменные должны выглядеть так:

```
matrix World;
matrix View;
matrix Projection;

float4 LightColor;
float4 LightPosition;
float4 LightDirection;
float  Phi;
float  Theta;
```

Функцию вершинного шейдера оставляем без изменений. То есть такой, какой она была в случае точечного источника света:

```
PS_INPUT VS( VS_INPUT Data )
{
  PS_INPUT Out;
  Out = (PS_INPUT)0;

  Out.Pos = mul (Data.Pos, World);

  Out.InterPos = Out.Pos.xyz;

  Out.Pos = mul (Out.Pos, View);
  Out.Pos = mul (Out.Pos, Projection);
```

```
  Out.Normal = mul (Data.Normal, World);

  return Out;
}
```

Дальше должна идти функция пиксельного шейдера. Чтобы ее упростить, давайте введем вспомогательную функцию, которая возвращает значение освещенности точки в зависимости от угла α (угол между направлением, в котором нацелен источник, и вектором от источника света к точке). В качестве параметров она будет принимать два вектора: направление, в котором нацелен прожектор, и вектор направления на источник света, выходящий из текущей точки. Объявляем функцию:

```
float GetAngleIntensity ( float3 LightDir, float3 PixelToLight )
```

В теле функции присваиваем значения показателю степени F и инициализируем переменную, в которую мы поместим полученное значение освещенности:

```
float F = 1.0;
float AngleIntensity = 0.0;
```

Далее, если следовать изложенной выше теории, мы должны найти угол α и с его помощью определить, в каком световом конусе находится точка. Так сделать, конечно, можно, но не рационально. В этом случае нам придется сделать много лишних вычислений. Лучше мы поступим по-другому. Введем переменную `CosAlpha`, в которой будет храниться косинус угла α, и все необходимые вычисления мы будем делать с его помощью и с помощью косинусов углов $\frac{\theta}{2}$ и $\frac{\varphi}{2}$. Косинус угла α вычисляется как скалярное произведение векторов, указанных в качестве входных параметров. В качестве параметра `PixelToLight` передается вектор направления из точки к источнику света, но нам нужен вектор, направленный наоборот, поэтому мы перед вторым вектором ставим минус:

```
float CosAlpha = saturate(dot(LightDir, -PixelToLight));
```

Далее мы сравниваем полученное значение с косинусами углов $\frac{\theta}{2}$ и $\frac{\varphi}{2}$, и в зависимости от результата устанавливаем значение освещенности:

```
if(CosAlpha>cos(Theta/2)) AngleIntensity=1.0;
  else if(CosAlpha<cos(Phi/2)) AngleIntensity=0.0;
         else AngleIntensity = saturate(pow((CosAlpha-cos(Phi/2))/
                                    (cos(Theta/2)-cos(Phi/2)),F));
```

Возвращаем рассчитанное значение и выходим из функции:

```
return AngleIntensity;
```

Вооружившись этой полезной функцией, написать функцию пиксельного шейдера будет совсем несложно. Итак, объявим функцию пиксельного шейдера:

```
float4 PS_SpotLight( PS_INPUT Vertex ) : SV_Target
```

В самом начале функции присваиваем значения параметрам, которыми регулируется затухание света в зависимости от расстояния:

```
float A = 0.2;
float B = 0.0;
float C = 0.0;
```

Вычисляем вектор, направленный из данной точки к источнику света:

```
float3 PixelToLight = (float3)LightPosition - Vertex.InterPos;
```

Нормализуем полученный вектор и вектор нормали к поверхности в данной точке:

```
float3 NewNormal = normalize(Vertex.Normal);
float3 NewDirection = normalize(PixelToLight);
```

Вычисляем затухание света в зависимости от расстояния:

```
float LightAtten =1/( A*pow(length(PixelToLight),2.0) +
                      B*length(PixelToLight)+C );
```

Вызываем вспомогательную функцию для расчета интенсивности света в зависимости от угла α. При этом нормализуем вектор направления, в котором нацелен прожектор, и приводим его к вектору из трех координат (нормализация необходима для получения правильного значения косинуса угла α):

```
float LightAngle =
    GetAngleIntensity (normalize((float3)LightDirection), NewDirection);
```

Все компоненты интенсивности света теперь известны, вычисляем общую интенсивность падающего света с учетом цвета лучей прожектора:

```
float4 LightIntensity = LightColor*LightAngle*LightAtten;
```

Зная интенсивность падающего света, вычисляем интенсивность отраженного света, то есть цвет пиксела:

```
float4 FinalColor = saturate( LightIntensity*
                              dot(NewNormal, NewDirection) );
FinalColor.a = 1;
```

Возвращаем полученный цвет пиксела и выходим из функции:

```
return FinalColor;
```

Сохраним все изменения, откомпилируем и запустим проект. На экране должно появиться изображение куба, на который падает луч света от прожектора (рис. 7.14).

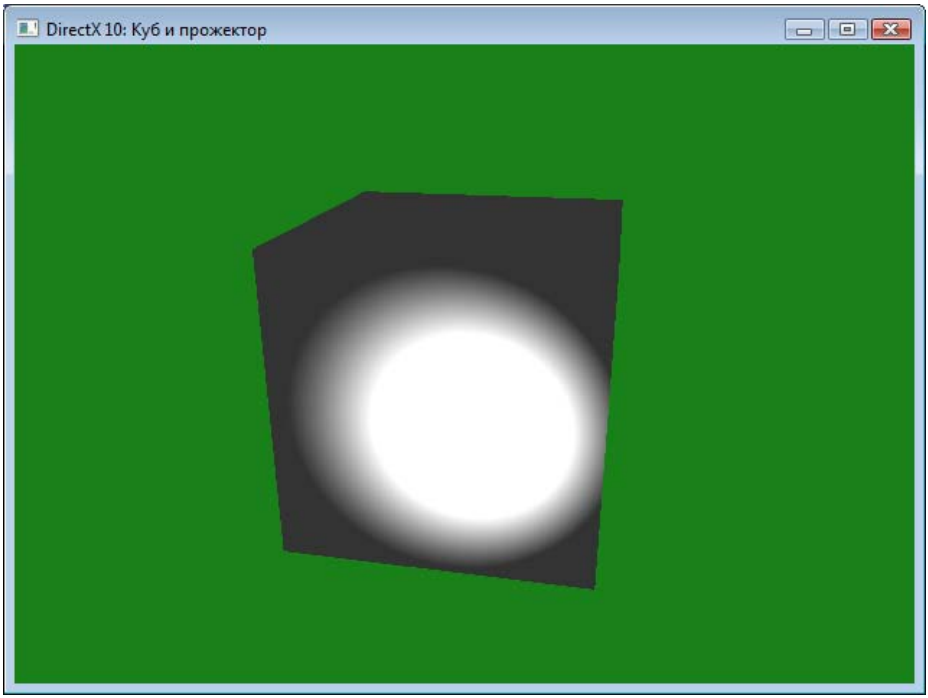

Рис. 7.14. Куб с освещением типа прожектор

Весь исходный текст файла "SpotLight.fx" представлен в листинге 7.2. Готовый проект и файл эффектов можно найти на компакт-диске в папке Glava7/SpotLight. Прежде чем мы отправимся дальше, хотелось бы прояснить еще один момент. На получившемся изображении области, куда не попадают лучи прожектора, выглядят абсолютно черными. Это и понятно: такова особенность этого источника света, что свет от него распространяется только в ограниченном объеме. Чтобы добавить какое-то количество света ко всем точкам сразу (равномерное освещение — ambient light), можно немного изменить строку, где рассчитывается цвет пиксела:

```
float4 FinalColor = saturate(0.2 +
            max(LightIntensity*dot(NewNormal, NewDirection),0) );
```

В этом варианте к рассчитанному значению добавляется константа 0.2, то есть области, куда не попадает свет, уже будут иметь не нулевую освещенность. Функция `max()` здесь предназначена для исключения использования отрицательных значений скалярного произведения. В данном случае отрицательное значение получается, если угол между падающим лучом и нормалью больше 90° (луч света падает с тыльной стороны грани).

Листинг 7.2

```
// Шейдеры для прожектора
// Файл SpotLight.fx
//--------------------------------------------------------------------------
// Структуры данных
//--------------------------------------------------------------------------
struct VS_INPUT
{
    float4 Pos : POSITION;
    float3 Normal : NORMAL;
};

struct PS_INPUT
{
    float4 Pos : SV_POSITION;
    float3 InterPos : TEXTCOORD0;
    float3 Normal : TEXTCOORD1;
};
//--------------------------------------------------------------------------
// Глобальные переменные
//--------------------------------------------------------------------------
matrix World;
matrix View;
matrix Projection;

float4 LightColor;
float4 LightPosition;
float4 LightDirection;
float  Phi;
float  Theta;

//--------------------------------------------------------------------------
// Функция вершинного шейдера
//--------------------------------------------------------------------------
PS_INPUT VS( VS_INPUT Data )
{
    PS_INPUT Out;
```

```
    Out = (PS_INPUT)0;

    // Умножим координаты вершины
    // на матрицу преобразований
    Out.Pos = mul (Data.Pos, World);

    // Подготовим эти координаты для
    // передачи в пиксельный шейдер
    Out.InterPos = Out.Pos.xyz;

    // Умножим координаты на
    // оставшиеся матрицы
    Out.Pos = mul (Out.Pos, View);
    Out.Pos = mul (Out.Pos, Projection);

    Out.Normal = mul (Data.Normal, World);

    return Out;
}

// Функция расчета интенсивности света
// в зависимости от угла альфа
float GetAngleIntensity ( float3 LightDir, float3 PixelToLight )
{
  float F = 1.0;
  float AngleIntensity = 0.0;

  float CosAlpha = saturate(dot(LightDir, -PixelToLight));

   if(CosAlpha>cos(Theta/2)) AngleIntensity=1.0;
   else if(CosAlpha<cos(Phi/2)) AngleIntensity=0.0;
   else AngleIntensity = saturate(pow((CosAlpha-cos(Phi/2))/
                                      (cos(Theta/2)-cos(Phi/2)),F));

   return AngleIntensity;
}

//--------------------------------------------------------------------------
// Функция пиксельного шейдера
//--------------------------------------------------------------------------
```

```
float4 PS_SpotLight( PS_INPUT Vertex ) : SV_Target
{
    float A = 0.2;
    float B = 0.0;
    float C = 0.0;

    float3 PixelToLight = (float3)LightPosition - Vertex.InterPos;

    float3 NewNormal = normalize(Vertex.Normal);
    float3 NewDirection = normalize(PixelToLight);

    float LightAtten =1/
        ( A*pow(length(PixelToLight),2.0)+B*length(PixelToLight)+C );

    float LightAngle = GetAngleIntensity (
                        normalize((float3)LightDirection), NewDirection);

    float4 LightIntensity = LightColor*LightAngle*LightAtten;

    float4 FinalColor = saturate(0.2+
        max(LightIntensity*dot(NewNormal, NewDirection),0) );
    FinalColor.a = 1;

    return FinalColor;
}
//--------------------------------------------------------------------------
// Техника отображения
//--------------------------------------------------------------------------
technique10 RenderSpotLight
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, VS() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, PS_SpotLight() ) );
    }
}
```

Материалы

Свойства самой поверхности, на которую падает свет, тоже играют большую роль в том, как они воспринимаются человеческим глазом. Например, есть два абсолютно одинаковых куба, находящиеся в одинаковых условиях освещения. Если один сделан из стали, а другой из пластмассы, они выглядят совершенно по-разному. В Direct3D под материалом понимают описание способности поверхности отражать или испускать свет. Свойства поверхности описывают через указание соответствующих цветовых компонентов:

- Цвет рассеянного света (diffuse color) — определяет, как отражается свет от поверхности. С его помощью задается цвет поверхности, когда на нее попадает свет.
- Цвет подсветки (ambient color) — определяет цвет поверхности, если свет на нее не попадает. Обычно значение этого параметра совпадает со значением для рассеянного света.
- Цвет испускаемого света (emissive color) — создает эффект свечения поверхности при попадании на нее света.
- Цвет зеркальных бликов (specular color) — с помощью этой составляющей задается цвет бликов, которые создают эффект блеска.

Немного мы уже прикоснулись к материалам, когда использовали цвет подсветки в примере с прожектором. В нашей модели освещения расчет данных параметров не предусмотрен. Разбор моделей освещения, выполняющих расчеты всех этих параметров, выходит за рамки нашей книги. Для углубленного изучения данной темы придется обратиться к специализированной литературе.

Текстуры

Мы можем добавить сколько угодно источников света к нашей трехмерной сцене, но все же не получим изображение, напоминающее реальный мир. У каждой поверхности из окружающих нас тел есть своя индивидуальность. На деревянной столешнице стола можно увидеть узор из древесных волокон, стену дома можно узнать по линиям кирпичной кладки, кора дерева содержит множество неровностей, также образующих неповторимый рисунок... Все это многообразие одним лишь освещением передать невозможно. Зато с помощью текстур это можно смоделировать очень просто. Что же такое текстура? Текстура — это изображение (картинка), которое накладывается на поверхность с целью сделать ее похожей на какой-либо объект из реального мира. Проще говоря, эффект от наложения текстуры на поверхность аналоги-

чен эффекту от наклейки фотообоев на стены комнаты: поверхности те же самые, а воспринимаются уже по-другому.

Для того чтобы наложить текстуру на объект, нам понадобится сама текстура (например, файл формата BMP), а также текстурные координаты, указанные для каждой вершины поверхности.

> ***Замечание***
>
> В Direct3D 10 поддерживаются графические файлы следующих форматов: BMP, JPG, PNG, DDS, TIFF, GIF, WMP. (Последний формат — файлы программы Windows Media Player.) Формат TGA больше не поддерживается по совершенно непонятным для меня причинам.

Текстурные координаты (они обозначаются как U и V) показывают, какая точка на текстуре должна проецироваться на данную вершину. Так как размеры текстуры, накладываемой на поверхность, в разных случаях могут быть разными, текстурные координаты изменяются в интервале от 0 до 1. На рис. 7.15 изображена грань куба с наложенной текстурой.

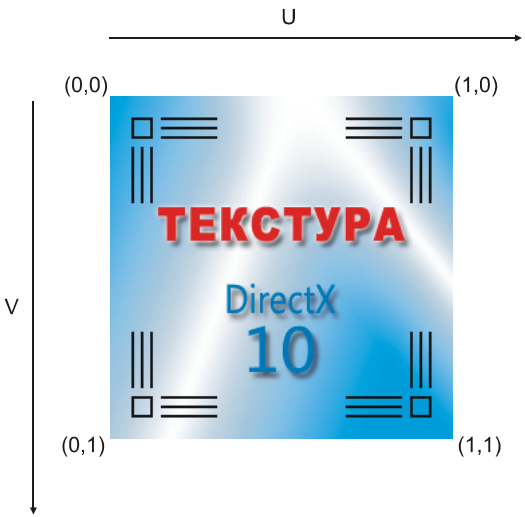

Рис. 7.15. Текстура и текстурные координаты

У каждой вершины обозначены текстурные координаты. Координаты (0,0) соответствуют верхнему левому углу текстуры, координаты (1,1) — нижнему правому углу. Первой указывается координата U, затем координата V. В случае наложения текстуры на квадратную грань, для каждой вершины грани указываются координаты соответствующего угла текстуры. Еще раз проследим по рисунку, каким вершинам соответствуют текстурные координаты (0,0), (1,0), (0,1), (1,1).

Режимы адресации текстуры

Текстурные координаты могут быть как больше, так и меньше единицы. Указание текстурных координат больше единицы обычно показывает, что в этом направлении текстуру необходимо повторить. Например, чтобы дважды повторить текстуру как вдоль оси U, так и вдоль оси V, для вершин грани следует задать текстурные координаты, как показано на рис. 7.16.

Существуют различные способы отображения (адресации) текстур при использовании текстурных координат, больших единицы, пример соответствует лишь одному из них. Что именно мы увидим за пределами единичного квадрата текстурных координат, зависит от установленного режима адресации. Рассмотрим, какие режимы предусмотрены в Direct3D 10.

Рис. 7.16. Пример использования текстурных координат со значениями больше единицы

Обертывание (wrap)

Текстура повторяется в неизменном виде. Заполнение грани текстурой аналогично использованию опции "Замостить" при установке рисунка рабочего стола. Пример наложения текстуры с применением данного режима адресации можно увидеть на рис. 7.17.

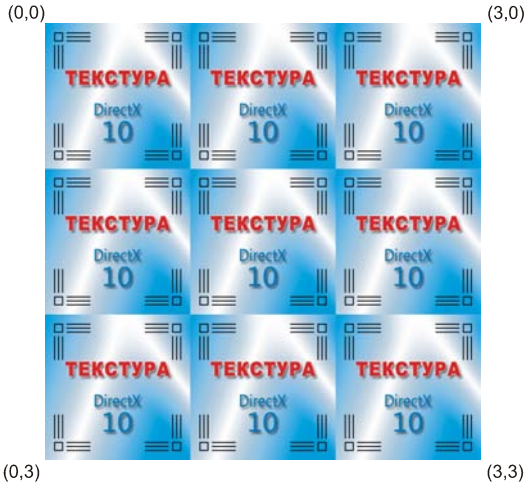

Рис. 7.17. Режим адресации текстуры "обертывание" (wrap)

Отражение (mirror)

При использовании этого режима текстура зеркально отражается при каждом новом повторении. Например, если координата U изменяется от 1.0 до 3.0, то текстура в интервале от 0.0 до 1.0 будет отображаться нормально, от 1.0 до 2.0 отображение буде производиться зеркально, в интервале от 2.0 до 3.0 текстура опять будет отображаться нормально. Наложение текстуры с помощью этого режима адресации представлено на рис. 7.18.

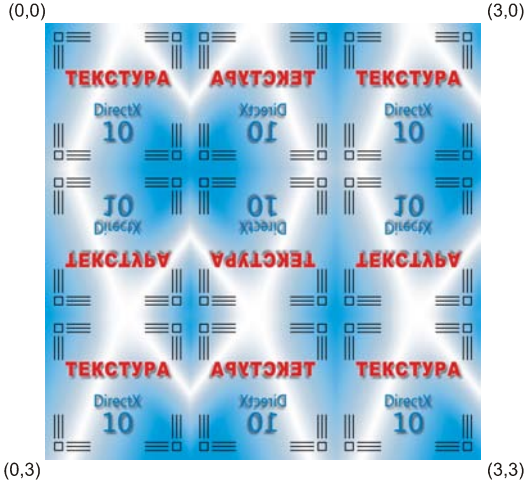

Рис. 7.18. Режим адресации текстуры "отражение" (mirror)

Фиксирование (clamp)

В интервале текстурных координат от 0.0 до 1.0 текстура отображается нормальным образом, после чего берется последняя строка и столбец пикселов текстуры и протягивается до границы примитива, на который текстура накладывается. Описанный способ графически представлен на рис. 7.19.

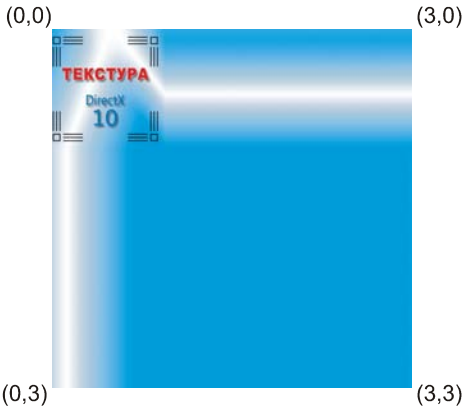

Рис. 7.19. Режим адресации текстуры "фиксация" (clamp)

Рамка (border)

При использовании этого режима текстура также отображается нормально в интервале текстурных координат от 0.0 до 1.0, далее происходит заполнение установленным однородным цветом. Пример наложения текстуры с использованием данного режима представлен на рис. 7.20.

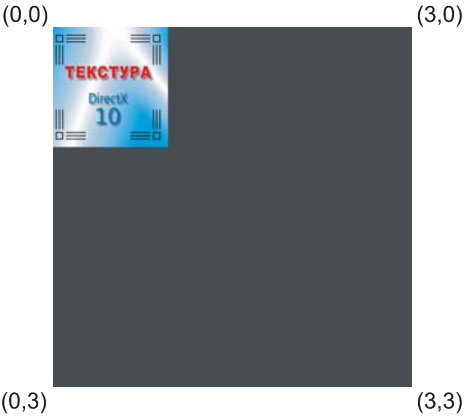

Рис. 7.20. Режим адресации текстуры "рамка" (border)

Фильтрация текстур

При наложении текстуры на грани трехмерного объекта возникают ситуации, когда, например, объект находится на близком расстоянии от камеры и текстуру для наложения приходится растягивать. Обратная ситуация возникает, когда объект удален от камеры, тогда текстуру приходится уменьшать. Фильтрацией текстур называется процесс вычисления цветов пикселов примитивов в соответствии с накладываемыми текстурами. При этом происходит либо уменьшение, либо увеличение текстуры. Увеличением текстуры называется ситуация, когда на один элемент текстуры (пикселы текстуры, как мы уже знаем, правильнее называть текселами) приходится несколько пикселов примитива. Уменьшением же, напротив, называется наложение нескольких текселов на один пиксел примитива.

В Direct3D 10 имеется около двадцати различных алгоритмов фильтрации текстур. Все мы рассматривать, конечно, не будем, рассмотрим только несколько примеров:

- MIN_MAG_MIP_POINT — в обозначении этого алгоритма указывается, что для увеличения (magnifying) и уменьшения (minifying) текстуры, а также для уровней мипмаппинга используется точечное сэмплирование. Это означает, что для получения цвета одного пиксела примитива используется один тексел.
- MIN_MAG_MIP_LINEAR — этот алгоритм для увеличения и уменьшения текстуры, а также для уровней мипмаппинга использует линейное сэмплирование. Это означает, что для получения цвета одного пиксела примитива используется среднее значение, полученное на основе двух элементов текстуры.

Теоретическая часть окончена. Не будем откладывать практику в долгий ящик и попытаемся наложить текстуру на наш вращающийся куб.

Наложение текстуры на куб

Приступим к составлению очередного шедевра нашей программистской мысли. Создадим новый проект и назовем его "CubeText". Скопируем в него код из примера с направленным источником света. Нам также потребуется создать в папке с проектом файл "CubeText.fx", в него мы, как обычно, запишем код шейдеров. Работы будет не очень много, так как процесс наложения текстур автоматизирован гораздо лучше, чем процесс вычисления освещения. Открываем проект и начинаем.

Как вы уже, наверное, догадались, для передачи текстурных координат нам предстоит скорректировать формат вершины. Его версия для координат вершины в пространстве и текстурных координат выглядит следующим образом:

```
struct SimpleVertex
{
  D3DXVECTOR3 Pos;
  D3DXVECTOR2 Texcoord;
};
```

Теперь перейдем к глобальным переменным, удалим ненужные переменные для источника света. Мы с вами уже работали с файлами изображений, когда выводили на экран спрайт. И мы знаем, что при этом требуется создать представление данных как ресурса шейдера. Для наложения текстуры нам также необходима переменная шейдера, через которую мы свяжем ресурс шейдера с графическим конвейером. Объявим переменную для представления данных:

```
ID3D10ShaderResourceView* g_pShaderResource = NULL;
```

И переменную для связи с шейдером:

```
ID3D10EffectShaderResourceVariable* g_pTexResource = NULL;
```

Вот так должно выглядеть объявление всех глобальных переменных в нашей программе:

```
HWND        g_hWnd = NULL;
D3D10_DRIVER_TYPE       g_driverType = D3D10_DRIVER_TYPE_NULL;
ID3D10Device*           g_pd3dDevice = NULL;
IDXGISwapChain*         g_pSwapChain = NULL;
ID3D10RenderTargetView* g_pRenderTargetView = NULL;

ID3D10Effect*           g_pEffect = NULL;
ID3D10EffectTechnique*  g_pTechnique = NULL;
ID3D10InputLayout*      g_pVertexLayout = NULL;
ID3D10Buffer*           g_pVertexBuffer = NULL;

ID3D10Buffer*               g_pIndexBuffer = NULL;

ID3D10EffectMatrixVariable* g_pWorldVariable = NULL;
ID3D10EffectMatrixVariable* g_pViewVariable = NULL;
ID3D10EffectMatrixVariable* g_pProjectionVariable = NULL;
ID3D10EffectShaderResourceVariable* g_pTexResource = NULL;

ID3D10ShaderResourceView* g_pShaderResource = NULL;
```

```
D3DXMATRIX                  g_World;
D3DXMATRIX                  g_View;
D3DXMATRIX                  g_Projection;
```

Готово. Теперь открываем функцию `InitDirect3D 10()`. Меняем имя загружаемого файла эффектов ("CubeText.fx") и исправляем имя извлекаемой техники отображения ("RenderTexture"). Далее мы должны создать представление данных как ресурса шейдера. Как и в примере с выводом на экран спрайта, мы воспользуемся функцией `D3DX10CreateShaderResourceViewFromFile()`. Можно воспользоваться соответствующим фрагментом кода, исправив лишь имя файла загружаемой текстуры на "dx10_tex.bmp":

```
D3DX10_IMAGE_INFO InfoFromFile;
D3DX10_IMAGE_LOAD_INFO LoadImageInfo;

hr = D3DX10GetImageInfoFromFile( L"dx10_tex.bmp", NULL,
                                 &InfoFromFile, NULL );

LoadImageInfo.Width = InfoFromFile.Width;
LoadImageInfo.Height = InfoFromFile.Height;
LoadImageInfo.Depth = InfoFromFile.Depth;
LoadImageInfo.FirstMipLevel = 1;
LoadImageInfo.MipLevels = InfoFromFile.MipLevels;
LoadImageInfo.Usage = D3D10_USAGE_DEFAULT;
LoadImageInfo.BindFlags = D3D10_BIND_SHADER_RESOURCE;
LoadImageInfo.CpuAccessFlags = 0;
LoadImageInfo.MiscFlags = 0;
LoadImageInfo.Format = InfoFromFile.Format;
LoadImageInfo.Filter = D3DX10_FILTER_NONE;
LoadImageInfo.MipFilter = D3DX10_FILTER_NONE;
LoadImageInfo.pSrcInfo = &InfoFromFile;

hr = D3DX10CreateShaderResourceViewFromFile( g_pd3dDevice,
       L"dx10_tex.bmp", &LoadImageInfo, NULL, &g_pShaderResource, NULL );
if( FAILED(hr) )
  return hr;
```

Далее свяжем ресурс шейдера с переменной шейдера, для этого сначала обозначим тип шейдерной переменной:

```
g_pTexResource = g_pEffect->
                 GetVariableByName( "g_txDiffuse" )->AsShaderResource();
```

После этого установим ее значение:

```
g_pTexResource->SetResource( g_pShaderResource );
```

Затем скорректируем формат входных данных:

```
D3D10_INPUT_ELEMENT_DESC layout[] =
{
 { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
    D3D10_INPUT_PER_VERTEX_DATA, 0 },
 { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 12,
    D3D10_INPUT_PER_VERTEX_DATA, 0 }
};
```

Находим строки, в которых содержатся значения координат вершин. Нам необходимо каждой вершине назначить текстурные координаты. Как это сделать? Очень просто! Смотрим на грань с видимой стороны и назначаем координаты (0.0,0.0) для левой верхней вершины. Например, вот так текстурные координаты выглядят для верхней грани куба:

- Вершина 1: (0.0, 1.0)
- Вершина 2: (1.0, 1.0)
- Вершина 3: (1.0, 0.0)
- Вершина 4: (0.0, 0.0)

Полностью заполненный массив принимает следующий вид (в комментариях указаны номера вершин куба):

```
SimpleVertex vertices[] =
{
 { D3DXVECTOR3( -1.0f,  1.0f, -1.0f ), D3DXVECTOR2(0.0f, 1.0f) },//1
 { D3DXVECTOR3(  1.0f,  1.0f, -1.0f ), D3DXVECTOR2(1.0f, 1.0f) },//2
 { D3DXVECTOR3(  1.0f,  1.0f,  1.0f ), D3DXVECTOR2(1.0f, 0.0f) },//3
 { D3DXVECTOR3( -1.0f,  1.0f,  1.0f ), D3DXVECTOR2(0.0f, 0.0f) },//4

 { D3DXVECTOR3( -1.0f, -1.0f, -1.0f ), D3DXVECTOR2(0.0f, 0.0f) },//5
 { D3DXVECTOR3(  1.0f, -1.0f, -1.0f ), D3DXVECTOR2(1.0f, 0.0f) },//6
 { D3DXVECTOR3(  1.0f, -1.0f,  1.0f ), D3DXVECTOR2(1.0f, 1.0f) },//7
 { D3DXVECTOR3( -1.0f, -1.0f,  1.0f ), D3DXVECTOR2(0.0f, 1.0f) },//8

 { D3DXVECTOR3( -1.0f, -1.0f,  1.0f ), D3DXVECTOR2(0.0f, 1.0f) },//8
 { D3DXVECTOR3( -1.0f, -1.0f, -1.0f ), D3DXVECTOR2(1.0f, 1.0f) },//5
 { D3DXVECTOR3( -1.0f,  1.0f, -1.0f ), D3DXVECTOR2(1.0f, 0.0f) },//1
 { D3DXVECTOR3( -1.0f,  1.0f,  1.0f ), D3DXVECTOR2(0.0f, 0.0f) },//4
```

```
 { D3DXVECTOR3(  1.0f, -1.0f,  1.0f ), D3DXVECTOR2(1.0f, 1.0f) },//7
 { D3DXVECTOR3(  1.0f, -1.0f, -1.0f ), D3DXVECTOR2(0.0f, 1.0f) },//6
 { D3DXVECTOR3(  1.0f,  1.0f, -1.0f ), D3DXVECTOR2(0.0f, 0.0f) },//2
 { D3DXVECTOR3(  1.0f,  1.0f,  1.0f ), D3DXVECTOR2(1.0f, 0.0f) },//3

 { D3DXVECTOR3( -1.0f, -1.0f, -1.0f ), D3DXVECTOR2(0.0f, 1.0f) },//5
 { D3DXVECTOR3(  1.0f, -1.0f, -1.0f ), D3DXVECTOR2(1.0f, 1.0f) },//6
 { D3DXVECTOR3(  1.0f,  1.0f, -1.0f ), D3DXVECTOR2(1.0f, 0.0f) },//2
 { D3DXVECTOR3( -1.0f,  1.0f, -1.0f ), D3DXVECTOR2(0.0f, 0.0f) },//1

 { D3DXVECTOR3( -1.0f, -1.0f,  1.0f ), D3DXVECTOR2(1.0f, 1.0f) },//8
 { D3DXVECTOR3(  1.0f, -1.0f,  1.0f ), D3DXVECTOR2(0.0f, 1.0f) },//7
 { D3DXVECTOR3(  1.0f,  1.0f,  1.0f ), D3DXVECTOR2(0.0f, 0.0f) },//3
 { D3DXVECTOR3( -1.0f,  1.0f,  1.0f ), D3DXVECTOR2(1.0f, 0.0f) },//4
};
```

Оставшаяся часть функции не меняется. Работа с исходным текстом основной программы закончена, переходим к составлению шейдера. В первую очередь, по традиции, определимся со структурами входных параметров для функций шейдеров:

```
struct VS_INPUT
{
  float4 Pos : POSITION;
  float2 Tex : TEXCOORD;
};

struct PS_INPUT
{
  float4 Pos : SV_POSITION;
  float2 Tex : TEXCOORD;
};
```

Какие-либо пояснения тут, наверное, излишни. Можно только отметить, что здесь мы используем семантику для текстурных координат по прямому назначению. Следующая структура нам пока не встречалась. Она описывает режим работы сэмплера, то есть способ фильтрации и отображения текстуры. Значения полей структуры устанавливают линейное сэмплирование и режим адресации текстуры "обертывание":

```
SamplerState samLinear
{
  Filter = MIN_MAG_MIP_LINEAR;
```

```
  AddressU = Wrap;
  AddressV = Wrap;
};
```

Аналогичную структуру мы использовали для установки режима растеризатора, когда нам требовалось отключить отсечение невидимых граней. В отличие от того случая, мы передадим данную структуру в качестве параметра методу, который будет вычислять цвет пиксела, но об этом речь впереди. Взглянем пока на список глобальных переменных:

```
matrix World;
matrix View;
matrix Projection;
Texture2D g_txDiffuse;
```

Последняя переменная в этом списке — как раз и есть тот текстурный объект, метод которого будет рассчитывать цвет пикселов. Переходим к составлению вершинного шейдера. Он должен умножить координаты вершины в пространстве на матрицы преобразований, а текстурные координаты передать в пиксельный шейдер без изменений. Текст вершинного шейдера выглядит таким образом:

```
PS_INPUT VS( VS_INPUT Data )
{
  PS_INPUT Out;
  Out = (PS_INPUT)0;

  Out.Pos = mul (Data.Pos, World);
  Out.Pos = mul (Out.Pos, View);
  Out.Pos = mul (Out.Pos, Projection);

  Out.Tex = Data.Tex;

  return Out;
}
```

Теперь переходим к пиксельному шейдеру. Он выглядит совсем просто:

```
float4 PS_Texture( PS_INPUT input ) : SV_Target
{
  return g_txDiffuse.Sample( samLinear, input.Tex );
}
```

Мы вызываем метод `sample` текстурного объекта HLSL. Этот метод требует указания двух параметров. Первым идет текстура с описанием режима работы сэмплера, а вторым — текстурные координаты, переданные из вершинного

шейдера. В качестве последнего штриха откорректируем имена функций шейдеров в описании техники отображения.

Наш проект готов к запуску. Скомпилируем и запустим его, в результате на экране должно появиться изображение, подобное представленному на рис. 7.21. Готовый проект и файл эффектов находятся на компакт-диске в папке Glava7/CubeText.

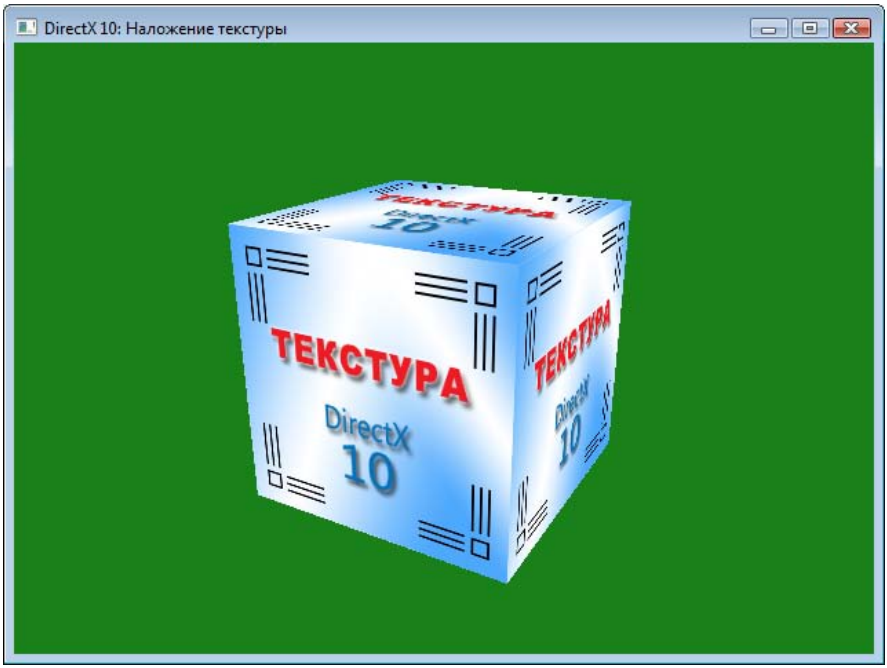

Рис. 7.21. Наложение текстуры на куб

К настоящему моменту мы с вами познакомились со всеми основными элементами, при помощи которых строится трехмерное изображение: вершины, нормали, источники света, текстуры. Вообще говоря, ничего принципиально нового они собой не представляют. То же освещение и наложение текстур было доступно и в предыдущих версиях Direct3D, с той лишь разницей, что работа с ними производилась несколько иначе. Осталось рассмотреть принципиальное нововведение, появившееся только в десятой версии Direct3D. О нем мы и поговорим в следующей главе.

Глава 8

Геометрические шейдеры

Особенности геометрического шейдера

Мы с вами уже довольно близко познакомились с вершинными и пиксельными шейдерами. Не рассмотрели мы только новый тип шейдеров, появившийся в десятой версии Direct3D — геометрические шейдеры. Наверное, из их названия не совсем понятно, чем геометрические шейдеры отличаются от вершинных и зачем они вообще понадобились.

Дело в том, что, например, вершинный шейдер в процессе работы получает одну вершину на входе и на выходе выдает тоже только одну, каким-то образом преобразованную вершину. Одно из отличий геометрического шейдера состоит в том, что он может обрабатывать сразу весь примитив целиком, то есть в качестве входного параметра может использоваться как вершина, так и отрезок (две вершины) либо треугольник (три вершины). Но самое главное отличие — это возможность создавать новую геометрию "на лету". Геометрический шейдер может получать на входе одну вершину и, основываясь на этой информации, генерировать на выходе много новых вершин. Это позволяет реализовать довольно интересные идеи, не привлекая при этом центральный процессор. До этого в Direct3D имелась возможность лишь видоизменять имеющийся набор данных, то есть просто масштабировать геометрию в сторону увеличения либо уменьшения размеров.

С точки зрения программирования, у функции геометрического шейдера имеются синтаксические отличия в объявлении от функций вершинных и пиксельных шейдеров:

- у входных параметров нужно дополнительно указывать тип примитива, который подается на вход шейдера;
- у выходных параметров нужно дополнительно указывать тип примитива, который шейдер выдает в качестве результата;
- в геометрическом шейдере должен указываться объем выходных данных.

Тип примитива у входных параметров

Наверное, будет понятнее, если сразу начать с примера, иллюстрирующего отличие в объявлении параметров. Нас пока интересуют исключительно входные параметры. Вот пример объявления функции:

```
GS_main( triangle float4 inputPos[3] )
{
  // Здесь код шейдера
}
```

Переведем на человеческий язык. Функция `GS_main()` в качестве параметра принимает массив `inputPos` из трех элементов типа `float4`, и эти элементы следует рассматривать как три вершины треугольника (`triangle`). Типы примитивов, которые поддерживаются геометрическими шейдерами, приведены в табл. 8.1.

Таблица 8.1. *Поддерживаемые типы примитивов для входных параметров*

Тип примитива	Описание
`point`	Список точек
`line`	Список либо последовательность отрезков
`triangle`	Список либо полоса треугольников
`lineadj`	Список либо последовательность отрезков с информацией о соседних элементах
`triangleadj`	Список либо полоса треугольников с информацией о соседних элементах

Компилятор принимает во внимание указание типа примитива только у функций верхнего уровня.

Тип примитива у выходных данных

С выводом данных ситуация несколько иная, к выходным данным предъявляются дополнительные требования. Кроме указания типа примитива, выходной параметр должен использовать потоковый объект (stream object) для вывода данных. Сам параметр с помощью ключевого слова `inout` обозначается как входной и выходной одновременно (входной, поскольку имеется возможность повторного направления результирующих данных на вход гео-

метрического шейдера). Пример объявления выходного параметра (пока отдельно от объявления функции):

```
inout TriangleStream<float3> outputPos
```

Переведем эту строку на наш с вами язык. Для ввода и вывода данных предлагается использовать потоковый объект `outputPos` данных типа `float3` и воспринимать последовательность выводимых значений как координаты вершин треугольников ("поток треугольников" — `TriangleStream`). В табл. 8.2 перечислены все типы примитивов для выходных данных, которые поддерживаются геометрическими шейдерами.

Таблица 8.2. *Поддерживаемые типы примитивов для выходных данных*

Тип примитива	Описание
`PointStream`	Вывод точек
`LineStream`	Вывод отрезков
`TriangleStream`	Вывод треугольников

У всех функций, которые играют роль геометрического шейдера, выходные параметры необходимо предварять ключевым словом `inout`, вне зависимости от того, является ли данная функция функцией верхнего уровня или нет.

Объем выходных данных

Под объемом выходных данных понимается максимальное количество вершин, которое может сгенерировать шейдер за время своего выполнения. Объем выходных данных указывается перед объявлением функции геометрического шейдера. Записывается это следующим образом:

```
[MaxVertexCount(n)]
```

где параметр `n` представляет собой максимальное количество вершин. Предельное количество 32-битных значений, которое может вывести геометрический шейдер, составляет 1024. Таким образом, если вершина описывается двумя 32-битными координатами, максимально возможное количество вершин составит 1024/2 = 512. В случае если для описания вершины используются три координаты, можно будет вывести уже только 341 вершину.

При достижении максимального количества сгенерированных вершин работа шейдера будет прекращена. Естественно, достижение этого максимума вовсе не является обязательным условием для завершения программы. Программист может предусмотреть выход из геометрического шейдера в любой нужный момент.

Пример объявления функции геометрического шейдера

Познакомившись со всеми элементами по отдельности, объединим их в единое целое. Рассмотрим полный пример объявления функции геометрического шейдера:

```
[MaxVertexCount(10)]
void GS_main( triangle float4 inputPos[3],
              inout TriangleStream<float3> outputPos )
{
  // Здесь код шейдера
}
```

Все это мы только что рассмотрели, и, скорее всего, пояснения излишни, но, на всякий случай, проговорим то, что закодировано в этой записи. Функция `GS_main()` использует два параметра. Первый, входной параметр `inputPos` — массив из трех элементов типа `float4`, который следует рассматривать как три вершины треугольника. Второй параметр, потоковый объект `outputPos`, выводит данные типа `float3`, эти значения представляет собой последовательность координат вершин треугольников. Параметр может использоваться как для вывода, так и для ввода данных. Функция `GS_main()` во время работы может вывести не более десяти вершин, в случае превышения этого количества работа функции прекращается.

Методы потоковых объектов

В теле геометрического шейдера используются те же самые операторы и внутренние функции HLSL, что и в других типах шейдеров. Но, поскольку геометрический шейдер использует для вывода данных потоковый объект, нужно использовать специальные методы этого объекта, которые этот вывод обеспечивают (табл. 8.3).

Таблица 8.3. *Методы потоковых объектов*

Метод потокового объекта	Тип потокового объекта	Описание
`Append(x)`	`PointStream`, `LineStream`, `TriangleStream`	Добавляет в поток данных параметр `x`. Тип параметра `x` должен соответствовать типу данных потокового объекта, указанному при описании выходного параметра

Таблица 8.3 *(окончание)*

Метод потокового объекта	Тип потокового объекта	Описание
`RestartStrip()`	`LineStream`, `TriangleStream`	Завершить текущую цепочку примитивов и начать новую. Если для завершения текущей цепочки недостаточно сгенерированных вершин, незаконченный примитив в конце цепочки будет отброшен

Считается, что метод `RestartStrip` начинает новую цепочку треугольников. Если необходимо выводить треугольники по отдельности (список треугольников), этот метод следует вызывать после вывода каждого треугольника. Веерный вывод треугольников не поддерживается.

Пишем геометрический шейдер

Для начала давайте определимся, что бы мы хотели получить от нашего геометрического шейдера. Конечно, было бы очень здорово, если бы он, получая на входе координаты одной вершины, на выходе генерировал бы сразу модель, скажем, вражеского звездолета. Но мы пока разбираемся с основами, и нам нужно что-нибудь попроще. Возьмем за основу пример с вращающимся кубом и напишем такой шейдер, который бы обеспечивал вывод на экран самого исходного куба, а также, дополнительно, копию каждой стороны куба, отодвинутую от оригинала на некоторое расстояние. Общая идея работы геометрического шейдера проиллюстрирована на рис. 8.1.

Как же нам все это реализовать? Начнем с самого начала. Договоримся, что в качестве входного параметра мы будем получать треугольники (тип примитива — `triangle`). Условимся также, что из приложения (и от вершинного шейдера к геометрическому) передаются координаты положения вершины и ее цвет. При этом алгоритм работы будет следующий:

1. Вывести в поток результатов треугольник, полученный в качестве входного параметра, без изменений (вывод исходной геометрии).
2. Вычислить вектор нормали к поверхности треугольника и в этом направлении на заданном расстоянии построить треугольник в параллельной плоскости (вывод новой геометрии). Чтобы выделить новую геометрию, цвет новых вершин установим белым.

В первом приближении наш геометрический шейдер уже можно себе представить. Для того чтобы превратить алгоритм в готовую программу, нам остается только выяснить, как рассчитать вектор нормали, имея координаты

вершин треугольника. Нам понадобятся два вектора, лежащие в плоскости треугольника, и результатом их векторного произведения, как мы с вами знаем, будет вектор, перпендикулярный к плоскости треугольника. Где взять эти вектора? Это просто: чтобы получить координаты вектора, направленного из точки A в точку B, нужно из координат конечной точки вычесть координаты начальной. Так как у нас вершины передаются как элементы массива, обозначим вершины треугольника согласно индексам их элементов. То есть в нашем случае мы будем рассматривать два вектора, которые имеют общее начало в точке 0 и заканчиваются в точках 1 и 2, соответственно (рис. 8.2).

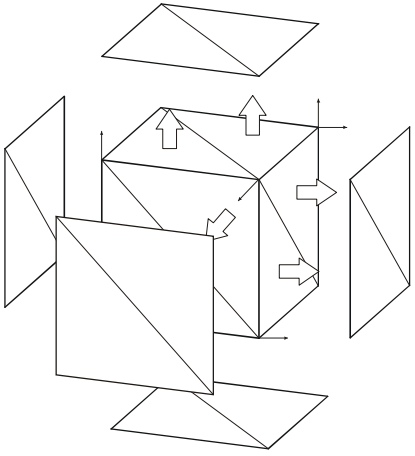

Рис. 8.1. Первый геометрический шейдер

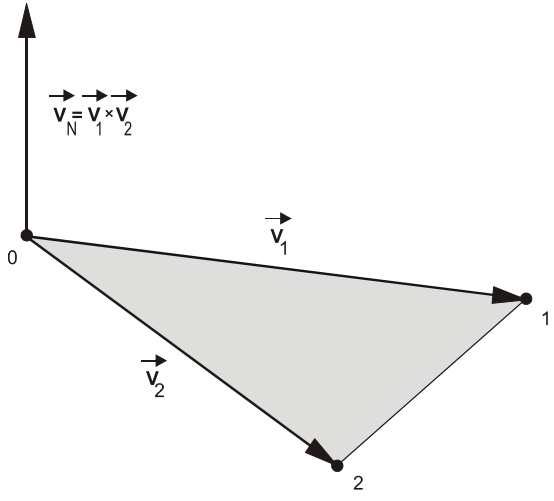

Рис. 8.2. Вычисление нормали к поверхности треугольника

Ну, вот теперь мы с теорией закончили, начинается практика. Итак, опишем структуры для хранения информации о вершинах:

```
struct VS_INPUT
{
    float4 Pos : POSITION;
    float4 Color: COLOR;
};
```

Эту структуру мы используем при передаче данных вершинному шейдеру. Нам понадобится еще одна структура:

```
struct GSPS_INPUT
{
    float4 Pos : SV_POSITION;
    float4 Color: COLOR;
};
```

Ее мы будем использовать при передаче данных геометрическому и пиксельному шейдеру. Теперь разберемся с вершинным шейдером, чтобы потом на него не отвлекаться. В этом примере он у нас совсем простой, выполняет умножение координат на мировую матрицу и копирует цвет вершины из входного параметра. Так как нам еще нужно будет генерировать новые вершины, умножение на видовую матрицу и матрицу проекции мы выполним позже, в геометрическом шейдере. Вот как выглядит код функции вершинного шейдера:

```
GSPS_INPUT VS( VS_INPUT Data )
{
    GSPS_INPUT Out;
    Out = (GSPS_INPUT)0;

    Out.Pos = mul (Data.Pos, World);
    Out.Color = Data.Color;
    return Out;
}
```

Приступаем непосредственно к функции геометрического шейдера. Вспомним, что мы договорились насчет типа примитивов входного параметра (`triangle`), поэтому логично будет выводить результаты в "поток треугольников" (`TriangleStream`). Теперь мы можем записать объявление функции нашего геометрического шейдера:

```
void GS( triangle GSPS_INPUT input[3],
 inout TriangleStream<GSPS_INPUT> TriStream )
```

Объем выходных данных мы пока не указываем, он нам пока неизвестен. Точное значение мы будем знать, когда наш шейдер будет закончен. Перейдем к телу шейдера. В первую очередь объявим временную переменную, в которой будут храниться данные для вывода (аналогично мы поступили в вершинном шейдере):

```
GSPS_INPUT output;
```

По алгоритму, который мы задумали реализовать, первым пунктом идет вывод исходной геометрии. В наш шейдер через параметр `input` передаются три вершины исходной геометрии (куба). При следующем выводе еще три, и так далее. Так как в вершинном шейдере необходимое преобразование координат мы выполнили не полностью (только умножение на мировую матрицу), умножение на матрицы вида и проекции мы выполним здесь, после чего и выведем три преобразованные вершины в поток выходных данных. Исходный текст, реализующий эти действия, такой:

```
for( int i=0; i<3; i++ )
 {
     output.Pos = input[i].Pos;
     output.Pos = mul( output.Pos, View );
     output.Pos = mul( output.Pos, Projection );
     output.Color = input[i].Color;

     TriStream.Append( output );
 }
```

В цикле производится умножение на матрицы и вывод в поток выходных данных с помощью метода `Append`. После цикла нам нужно вызвать метод `RestartStrip`, так как мы выводим не цепочку треугольников, а треугольники по отдельности. Первый шаг алгоритма мы реализовали. Остается еще второй, он будет немного сложнее. Начнем с получения вектора нормали к плоскости переданного в шейдер треугольника. Как это делается геометрически, мы уже знаем, а вот так это выглядит на языке HLSL.

```
float3 faceEdgeA = input[1].Pos - input[0].Pos;
float3 faceEdgeB = input[2].Pos - input[0].Pos;
float3 faceNormal = normalize( cross(faceEdgeA, faceEdgeB) );
```

Теперь в переменной `faceNormal` у нас содержится единичный вектор нормали. Чтобы получить вершину смещенного треугольника, нужно прибавить координаты вектора нормали к координатам вершины исходного треугольника. В этом случае получим смещение на единицу. Для получения любого другого смещения, единичный вектор нужно просто предварительно умножить на требуемое значение. Вывод новой вершины во многом похож на вывод исход-

ной геометрии: такой же цикл для обработки трех вершин, умножение на матрицы, вывод в поток. Разница в строках, где мы получаем координаты новой вершины:

```
output.Pos = input[i].Pos + float4(1.2*faceNormal, 0);
```

Как видно, здесь задается смещение 1.2 от исходного треугольника.

> ***Замечание***
>
> В этой строке мы можем наблюдать приведение типа результата выражения к типу `float4`. Выражение `float4(1.2*faceNormal,0)` означает, что создается вектор типа `float4`, три первые элемента которого будут равны трем элементам вектора `faceNormal`, умноженным на 1.2, а четвертый элемент равен нулю.

В следующей строке мы устанавливаем для новой вершины белый цвет:

```
output.Color = float4( 1.0f, 1.0f, 1.0f, 1.0f );
```

После цикла мы снова вызываем метод `RestartStrip`.

Ну, вот, дело сделано. Мы написали наш первый геометрический шейдер. Осталось только указать количество генерируемых вершин. Сейчас мы сделаем это с легкостью. Посмотрим, сколько раз у нас вызывается метод `Append`: три раза в первом цикле и три раза во втором — всего шесть раз. Это и есть количество генерируемых шейдером вершин. Значит, перед объявлением функции `GS` мы указываем именно это значение:

```
[MaxVertexCount(6)]
```

У нас остался не разобранным еще пиксельный шейдер, но он лишь возвращает переданный на вход цвет:

```
return input.Color;
```

Шаблонный буфер глубины

Шейдеры мы написали. Если сейчас запустить проект, то работать они, конечно, будут, но объекты при рисовании будут накладываться друг на друга. Дело в том, что до сих пор мы имели дело только с одним объектом в пространстве, и у нас не возникало ситуации, когда требуется отобразить один объект, частично закрытый другим объектом. Чтобы правильно решить такую задачу, компьютер должен знать, пикселы какого объекта находятся к камере ближе всего. В этом ему поможет *шаблонный буфер глубины*, который мы сейчас введем в основную программу. Этот буфер во многом похож на буфер визуализации, и в программе он используется аналогичным образом. Как ясно из названия, буфер состоит из двух частей: буфера глубины и шаблонного буфера. Если с буфером глубины более-менее все ясно, то

шаблонный буфер пока остается для нас загадкой. Проясним ситуацию. Шаблонный буфер используется в том случае, если нужно спрятать (не выводить) какие-то пикселы итогового изображения. Иначе говоря, шаблонный буфер разрешает или запрещает вывод определенных пикселов в буфер визуализации. Зачем такое могло понадобиться? В качестве примера возьмем уже упоминавшуюся игру, гонки на машинах. Как вывести изображение в зеркало заднего вида? Здесь может найти применение шаблонный буфер. Сначала выводится первая трехмерная сцена, то есть вид из глаз водителя, при этом с помощью шаблонного буфера запрещается вывод в ту область, где должно находиться зеркало заднего вида. После этого, на оставшийся чистым участок выводится изображение второй трехмерной сцены, то есть той же самой, но где камера направлена в другом направлении, так, чтобы получить соответствующее изображение в зеркале.

Шаблонный буфер глубины представляет собой двухмерную текстуру. Чтобы его использовать, мы должны выполнить такие действия:

1. Создать текстуру с нужными параметрами при помощи метода `ID3D10Device::CreateTexture2D`.
2. Для этой текстуры создать представление данных как шаблонного буфера глубины, используя метод `ID3D10Device::CreateDepthStencilView`.
3. Связать буфер глубины с графическим конвейером. Это делается с помощью метода `ID3D10Device::OMSetRenderTargets`, который мы пока используем только для привязки к конвейеру буфера визуализации.

Начнем выполнять этот план. Нам понадобятся две новые глобальные переменные:

```
ID3D10Texture2D*            g_pDepthStencil = NULL;
ID3D10DepthStencilView*     g_pDepthStencilView = NULL;
```

После добавления этих строк переходим к функции `InitDirect3D10()`. Поместим строки, отвечающие за создание шаблонного буфера глубины, после создания буфера визуализации. Сначала мы должны создать текстуру, для этого нам нужна заполненная структура типа `D3D10_TEXTURE2D_DESC`, описывающая ее параметры. Объявим соответствующую переменную:

```
D3D10_TEXTURE2D_DESC descDepth;
```

Заполним структуру, по ходу дела знакомясь с ее полями. Установим размеры создаваемой текстуры, ее ширину и высоту. Используем те же самые значения, что и для буфера визуализации:

```
descDepth.Width = width;
descDepth.Height = height;
```

Зададим количество уровней мипмаппинга, в нашем случае этот параметр равен 1:

```
descDepth.MipLevels = 1;
```

Определим количество элементов в массиве текстур. Нам нужна только одна текстура, поэтому присваиваем соответствующему полю значение 1:

```
descDepth.ArraySize = 1;
```

После этого установим формат текстуры, для нашего буфера нам подойдет формат `DXGI_FORMAT_D32_FLOAT`, он определяет каждый элемент текстуры как обычное 32-битное число:

```
descDepth.Format = DXGI_FORMAT_D32_FLOAT;
```

Следующие строки задают параметры мультисэмплинга. Мы его не используем, поэтому поставим параметры по умолчанию (означающие, что сглаживание не используется):

```
descDepth.SampleDesc.Count = 1;
descDepth.SampleDesc.Quality = 0;
```

Определим режим использования текстуры, то есть как будут считываться и записываться ее элементы. Здесь мы также воспользуемся значениями по умолчанию:

```
descDepth.Usage = D3D10_USAGE_DEFAULT;
```

Далее следует флаг, показывающий, в каком качестве текстура связывается с графическим конвейером. Для шаблонного буфера глубины укажем флаг `D3D10_BIND_DEPTH_STENCIL`:

```
descDepth.BindFlags = D3D10_BIND_DEPTH_STENCIL;
```

Последние два поля отвечают за режим доступа к текстуре центрального процессора и некоторые дополнительные параметры. Доступ для центрального процессора нам не нужен, дополнительные параметры тоже, поэтому везде укажем нули:

```
descDepth.CPUAccessFlags = 0;
descDepth.MiscFlags = 0;
```

На этом заполнение структуры закончилось. Вызовем метод для создания текстуры:

```
hr = g_pd3dDevice->CreateTexture2D( &descDepth, NULL, &g_pDepthStencil );
    if( FAILED(hr) )
        return hr;
```

В качестве параметров методу передаются указатель на заполненную структуру с параметрами текстуры, указатель на данные, которыми инициализируется текстура (для шаблонного буфера глубины мы эту возможность не ис-

пользуем), а также адрес указателя на созданную текстуру, который будет установлен методом.

Теперь нам нужно создать представление данных как шаблонного буфера глубины, для этого также необходимо заполнить структуру типа `D3D10_DEPTH_STENCIL_VIEW_DESC` для описания параметров. Она совсем небольшая. Объявляем переменную:

```
D3D10_DEPTH_STENCIL_VIEW_DESC descDSV;
```

Указываем формат данных ресурса, для которого создается представление данных. Копируем его из описания параметров текстуры:

```
descDSV.Format = descDepth.Format;
```

После этого нужно задать, что собой представляет ресурс. У нас это двухмерная текстура, используем соответствующее значение:

```
descDSV.ViewDimension = D3D10_DSV_DIMENSION_TEXTURE2D;
```

Для двухмерной текстуры устанавливаем, какой уровень мипмаппинга использовать. Укажем значение 0 для первого уровня с самым высоким разрешением (текстуру мы создали только с одним уровнем).

```
descDSV.Texture2D.MipSlice = 0;
```

Структуру мы заполнили, все параметры задали. Создаем представление данных:

```
hr = g_pd3dDevice->CreateDepthStencilView( g_pDepthStencil,
                                           &descDSV, &g_pDepthStencilView );
    if( FAILED(hr) )
        return hr;
```

В параметрах метода мы передаем указатель на содержащий данные ресурс (у нас это текстура), указатель на описание параметров представления данных и адрес указателя, который будет установлен на созданное представление данных. Можно было бы и не заполнять структуру, описывающую параметры, а при вызове метода вторым параметром указать `NULL`. В этом случае мы бы получили то же самое: представление данных, имеющее доступ к указанной текстуре на первом уровне мипмаппинга и использующее формат, с которым текстура создавалась. Так, конечно, проще, но зато мы узнали, как можно повлиять на параметры представления данных для шаблонного буфера глубины.

Созданное представление данных нужно связать с графическим конвейером. Сейчас у нас с графическим конвейером связывается только буфер визуализации:

```
g_pd3dDevice->OMSetRenderTargets( 1, &g_pRenderTargetView, NULL );
```

Чтобы связать еще и буфер глубины, нам нужно добавить адрес указателя на него в третьем параметре метода:

```
g_pd3dDevice->OMSetRenderTargets( 1, &g_pRenderTargetView,
                                  &g_pDepthStencilView );
```

Новая версия функции `InitDirect3D10()` готова, буфер глубины мы создали. Но буфер глубины необходимо очищать перед рисованием нового кадра, аналогично тому, как мы очищаем буфер визуализации. Сделаем это в функции `RenderScene()`, сразу после очистки буфера визуализации. Очистка буфера глубины проводится с помощью метода `ID3D10Device::ClearDepthStencilView`. Прототип этого метода выглядит следующим образом:

```
void ClearDepthStencilView(
  ID3D10DepthStencilView *pDepthStencilView,
  UINT ClearFlags,
  FLOAT Depth,
  UINT8 Stencil
);
```

Для вызова метода требуется указать следующие параметры:

- `pDepthStencilView` — указатель на очищаемый шаблонный буфер глубины;
- `ClearFlags` — флаги, определяющие, какую часть буфера очищать (мы используем флаги для очистки обеих частей буфера);
- `Depth` — значение, которым заполняется буфер глубины;
- `Stencil` — значение, которым заполняется буфер шаблона.

Сам вызов метода выглядит у нас таким образом:

```
g_pd3dDevice->ClearDepthStencilView( g_pDepthStencilView,
                 D3D10_CLEAR_DEPTH | D3D10_CLEAR_STENCIL , 1.0f, 0 );
```

Теперь нам осталось добавить в функцию `Cleanup()` освобождение памяти для представления данных как шаблонного буфера глубины:

```
if( g_pDepthStencilView ) g_pDepthStencilView->Release();
```

Полностью текст файла эффектов с геометрическим шейдером приведен в листинге 8.1. Его и соответствующий проект можно найти на прилагаемом компакт-диске в папке Glava8\GeomShader. Думаю, что если еще остались какие-то вопросы, то они должны исчезнуть после изучения листинга. Единственное, на что еще хочется обратить внимание, это на описание техники отображения. В описании нужно указать правильные параметры для установки геометрического шейдера:

```
SetGeometryShader( CompileShader( gs_4_0, GS() ) );
```

В технике отображения также используется режим растеризатора с отключенным отсечением невидимых граней, чтобы новые треугольники были видимы с обеих сторон.

Запустим наше приложение, чтобы увидеть геометрический шейдер в работе. Окно приложения с результатами его работы можно увидеть на рис. 8.3.

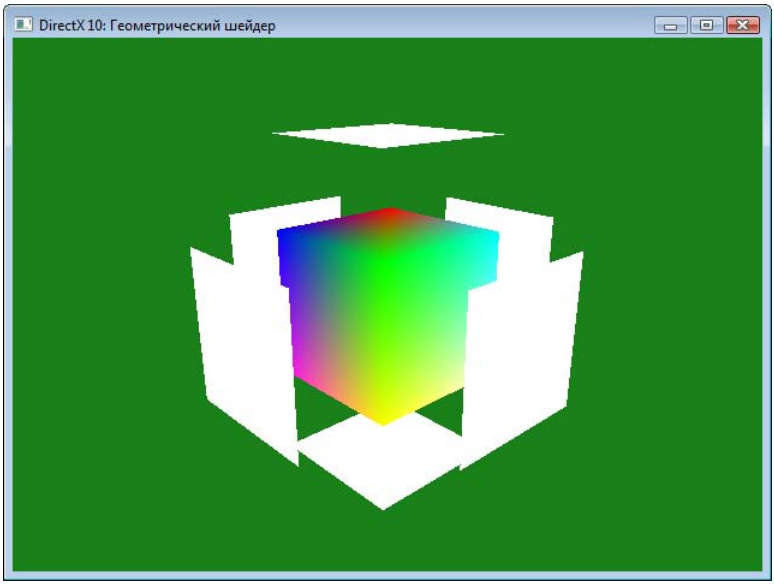

Рис. 8.3. Результат работы геометрического шейдера

Листинг 8.1

```
// Пример геометрического шейдера
// Файл cube_gs.fx
//--------------------------------------------------------------------------
// Структуры данных
//--------------------------------------------------------------------------
struct VS_INPUT
{
    float4 Pos : POSITION;
    float4 Color: COLOR;
};

struct GSPS_INPUT
```

```
{
    float4 Pos : SV_POSITION;
    float4 Color: COLOR;
};
//--------------------------------------------------------------------------
// Глобальные переменные
//--------------------------------------------------------------------------
matrix World;
matrix View;
matrix Projection;
//--------------------------------------------------------------------------
// Функция вершинного шейдера
//--------------------------------------------------------------------------
GSPS_INPUT VS( VS_INPUT Data )
{
    GSPS_INPUT Out;
    Out = (GSPS_INPUT)0;

    // Умножаем координаты вершины
    // на матрицу преобразования
    Out.Pos = mul (Data.Pos, World);

    // Передаем цвет без изменений
    Out.Color = Data.Color;
    return Out;
}

//--------------------------------------------------------------------------
// Функция геометрического шейдера
//--------------------------------------------------------------------------
[MaxVertexCount(6)]
void GS( triangle GSPS_INPUT input[3],
         inout TriangleStream<GSPS_INPUT> TriStream )
{
    GSPS_INPUT output;
    // Выводим куб
    for( int i=0; i<3; i++ )
    {
        output.Pos = input[i].Pos;
```

```
        output.Pos = mul( output.Pos, View );
        output.Pos = mul( output.Pos, Projection );

        output.Color = input[i].Color;

        TriStream.Append( output );
    }

    TriStream.RestartStrip();

    // Создаем новые элементы
    // Вычисляем нормаль к плоскости треугольника
    float3 faceEdgeA = input[1].Pos - input[0].Pos;
    float3 faceEdgeB = input[2].Pos - input[0].Pos;
    float3 faceNormal = normalize( cross(faceEdgeA, faceEdgeB) );

    for( int i=0; i<3; i++ )
    {
        output.Pos = input[i].Pos + float4(1.2*faceNormal,0);
        output.Pos = mul( output.Pos, View );
        output.Pos = mul( output.Pos, Projection );

        output.Color = float4( 1.0f, 1.0f, 1.0f, 1.0f );

        TriStream.Append( output );
    }

    TriStream.RestartStrip();
}
//--------------------------------------------------------------------------
// Функция пиксельного шейдера
//--------------------------------------------------------------------------
float4 PS_Color( GSPS_INPUT input ) : SV_Target
{
    return input.Color;
}
//--------------------------------------------------------------------------
// Техника отображения
//--------------------------------------------------------------------------
```

```
    // Создаем набор настроек растеризатора
    // без отсечения невидимых граней
    RasterizerState rsNoCulling { CullMode = None; };

technique10 RenderColorGS
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, VS() ) );
        SetGeometryShader( CompileShader( gs_4_0, GS() ) );
        SetPixelShader( CompileShader( ps_4_0, PS_Color() ) );

        SetRasterizerState ( rsNoCulling );
    }
}
```

На этом первая часть книги заканчивается. Мы уже умеем очень многое: печатать на экране текст, выводить картинки из файлов, самостоятельно создавать простые трехмерные объекты и накладывать на них текстуры, писать шейдеры для различных источников освещения… Я надеюсь, все это вдохновило вас на дальнейшее изучение трехмерной графики. Нас ждет вторая часть книги, в которой мы увидим программирование графики несколько с другой стороны. Мы будем учиться использовать каркас DXUT, при помощи которого мы можем существенно расширить свои возможности.

ЧАСТЬ II

ИСПОЛЬЗОВАНИЕ DXUT 10

Глава 9

Введение в DXUT

Что такое DXUT

Давайте вспомним, какая подготовка потребовалась, чтобы мы занялись непосредственно трехмерной графикой. Сначала нам нужно было разобраться с тем, как создать окно приложения. Для этого пришлось изучить структуру, которая содержит параметры окна и значения параметров, которые в ней содержатся. После этого мы выясняли, как создать устройство Direct3D 10, для этого тоже пришлось познакомиться с объемной структурой и ее полями. А сколько других функций пришлось рассмотреть за этот ознакомительный период? То есть, если начинать писать программу "с нуля", приходится затратить приличное время именно на подготовительный процесс, а не на саму графику. Это довольно неприятное открытие, учитывая тот факт, что времени всегда не хватает. Тем не менее, эта проблема решается довольно просто, если использовать DXUT.

Что такое DXUT? DXUT — сокращение от DirectX Utility (каркас DirectX) — представляет собой готовое минимальное приложение, использующее Direct3D. DXUT содержит множество полезных функций и классов для простого и быстрого доступа к возможностям DirectX. Чтобы убедиться в этом, сразу же перейдем к практике, тем более что на этот раз длительной подготовки не потребуется.

Первый проект с DXUT

Создадим первый проект с поддержкой DXUT. Сложного здесь ничего нет: запускаем программу DirectX Sample Browser. В открывшемся окне (рис. 9.1) отображается список программ-примеров, поставляющихся вместе с DirectX SDK.

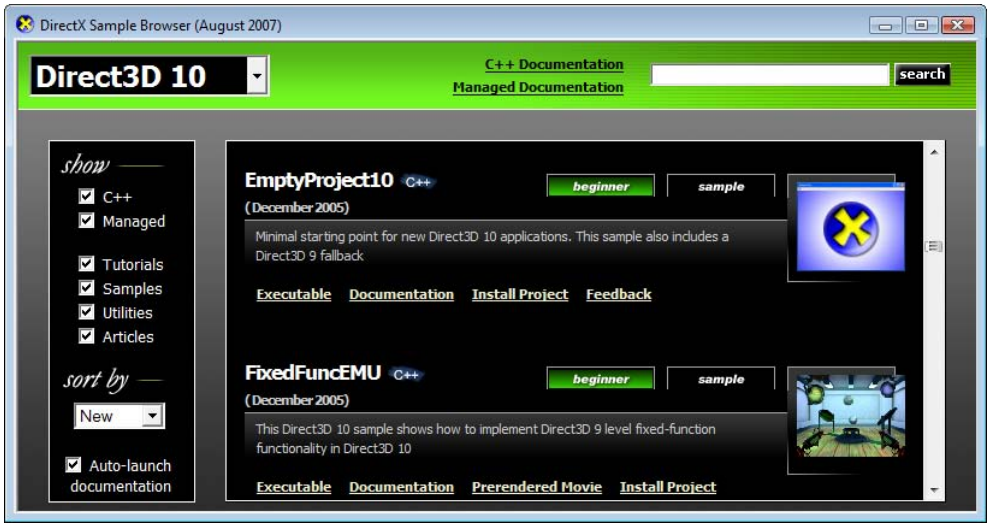

Рис. 9.1. Создание минимального проекта DXUT

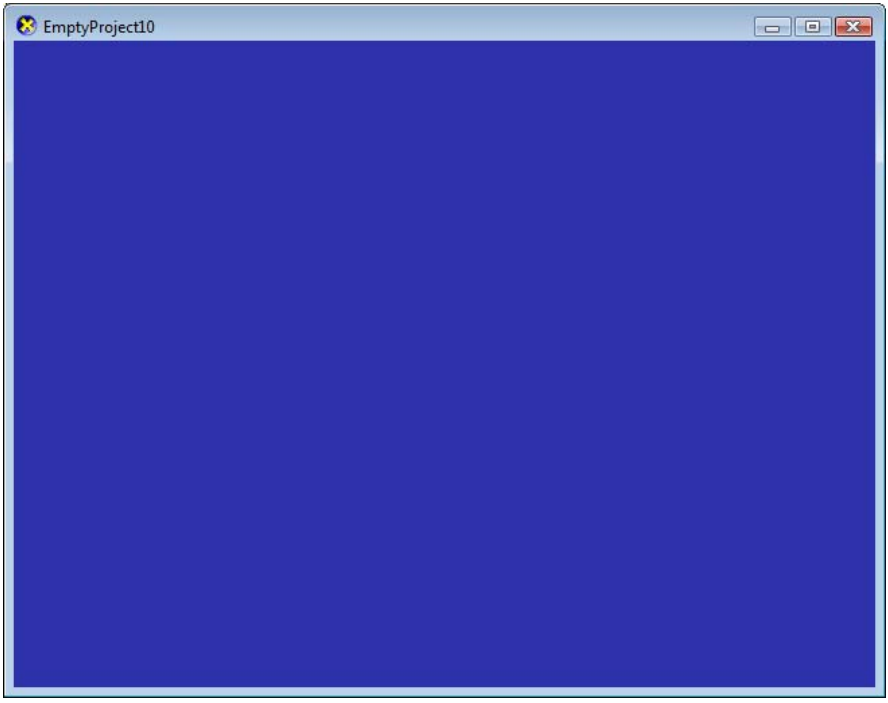

Рис. 9.2. Окно минимального проекта DXUT

Нас интересует пункт, озаглавленный "EmptyProject10". Выбираем Install Project (Установить проект) и указываем имя, под которым нужно скопировать проект и папку, куда его поместить. Откроем созданный проект, пробуем скомпилировать и запустить его. Ничего сверхъестественного не произойдет: появится окно с темно-синим фоном и надписью EmptyProject10 в заголовке (рис. 9.2).

У вас, наверное, сразу возник вопрос, какая необходимость привлекать какой-то там каркас, если мы сами уже написали такую программу, не проще ли использовать ее? Ответ на него такой: внешняя простота обманчива. Попробуйте нажать <Alt>+<Enter>, программа переключится в полноэкранный режим. Таким же образом переключимся обратно в оконный режим, нажмем клавишу <ESC> — программа закроется. Вот так, еще не написав ни строчки, мы уже получили возможность полноценного переключения в полноэкранный режим (с соответствующей обработкой события `WM_SIZE`, которую мы раньше осуществляли вручную), а также выход из программы по нажатию "горячей клавиши". Приятно, не правда ли?

Что у программы "снаружи", мы увидели, давайте теперь посмотрим, что у нее внутри. Откроем окно **Class View** (Просмотр функций/классов). Выберем в списке пункт **Global Functions and Variables** (Просмотр глобальных функций и переменных), как показано на рис. 9.3.

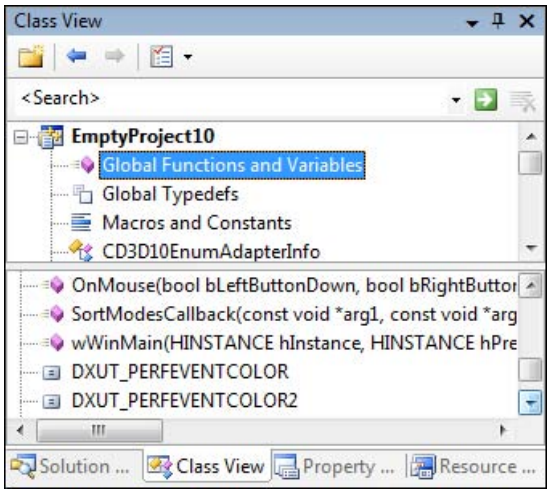

Рис. 9.3. Переключение на просмотр функций и глобальных переменных

Мы увидим много-много странных и непонятных нам функций. Правда, среди них все же есть одна, с которой мы знакомы. Да, правильно, — это функ-

ция `wWinMain()`. Только зачем ей еще одна буква "w" впереди? Все дело в том, что DXUT поддерживает полноценную работу с символами Unicode, и эта дополнительная буква в названии по существу показывает, что командная строка содержит символы Unicode, а не ANSI.

Вот как выглядит эта функция, с которой начинается выполнение программы (все комментарии переведены на русский язык):

```
int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPWSTR lpCmdLine, int nCmdShow)
{
 // Включить проверку памяти во время выполнения программы
 // для отладочных версий
#if defined(DEBUG) | defined(_DEBUG)
 _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif

 // DXUT создаст и будет использовать наилучшее из устройств
 // (D3D или D3D10),
 // в зависимости от того, какие функции из приведенных ниже
 // определены в системе

 // Установить общие функции обратного вызова
 DXUTSetCallbackFrameMove(OnFrameMove);
 DXUTSetCallbackKeyboard(OnKeyboard);
 DXUTSetCallbackMouse(OnMouse);
 DXUTSetCallbackMsgProc(MsgProc);
 DXUTSetCallbackDeviceChanging(ModifyDeviceSettings);
 DXUTSetCallbackDeviceRemoved(OnDeviceRemoved);

 // Настроить функции обратного вызова DXUT для варианта с
 // использованием D3D9.
 // Уберите эти строки, если использование D3D9 в программе
 // не требуется.
 DXUTSetCallbackD3D9DeviceAcceptable(IsD3D9DeviceAcceptable);
 DXUTSetCallbackD3D9DeviceCreated(OnD3D9CreateDevice);
 DXUTSetCallbackD3D9DeviceReset(OnD3D9ResetDevice);
 DXUTSetCallbackD3D9FrameRender(OnD3D9FrameRender);
 DXUTSetCallbackD3D9DeviceLost(OnD3D9LostDevice);
 DXUTSetCallbackD3D9DeviceDestroyed(OnD3D9DestroyDevice);
```

```
    // Настроить функции обратного вызова DXUT для варианта
    // с использованием D3D10.
    // Уберите эти строки, если использование D3D10 в программе
    // не требуется.
    DXUTSetCallbackD3D10DeviceAcceptable(IsD3D10DeviceAcceptable);
    DXUTSetCallbackD3D10DeviceCreated(OnD3D10CreateDevice);
    DXUTSetCallbackD3D10SwapChainResized(OnD3D10ResizedSwapChain);
    DXUTSetCallbackD3D10FrameRender(OnD3D10FrameRender);
    DXUTSetCallbackD3D10SwapChainReleasing(OnD3D10ReleasingSwapChain);
    DXUTSetCallbackD3D10DeviceDestroyed(OnD3D10DestroyDevice);

    // Здесь выполняется инициализация приложения

    // Разбор командной строки, вывод сообщений об ошибках,
    // дополнительные параметры командной строки не рассматриваются
    DXUTInit(true, true, NULL);
    // Отображение курсора и ограничение его передвижения в
    // полноэкранном режиме
    DXUTSetCursorSettings(true, true);
    DXUTCreateWindow(L"EmptyProject10");
    DXUTCreateDevice(true, 640, 480);
    // Вход в главный цикл визуализации DXUT

    DXUTMainLoop();

    // Здесь выполняется освобождение ресурсов приложения

    return DXUTGetExitCode();
}
```

Потихоньку начнем разбираться. Вся работа каркаса построена на функциях обратного вызова. Помните, мы с ними уже встречались? Разница здесь в том, что эти функции вызываются не операционной системой, а самим каркасом при реакции на какое-то событие. Но вернемся к функции `wWinMain()`. Первым как раз следует блок настройки функций обратного вызова. Проще говоря, определяется, какую функцию запускать при наступлении того или иного события. Как вы можете заметить, присутствуют версии функций как для Direct3D 9, так и для его десятой версии. Работа как с теми, так и с другими функциями осуществляется аналогичным образом, но наша с вами задача — освоить Direct3D 10, значит, и концентрироваться мы будем на функ-

циях, которые связаны именно с ним. Попробуем по названиям функций догадаться, за что они отвечают. Пожалуй, самое очевидное назначение у функции `OnD3D10FrameRender()`: в ней выполняется прорисовка сцены с помощью Direct3D 10. С остальными все уже не так просто, но здесь я помогу. Функция `OnD3D10CreateDevice()` осуществляет действия, которые нужно выполнить после создания устройства Direct3D 10. Назначение этой функции аналогично той части функции `InitDirect3D10()` из наших предыдущих проектов, которая следует за созданием устройства. Точно таким же образом функция `OnD3D10DestroyDevice()` производит действия, которые должны выполняться перед освобождением устройства Direct3D 10. В этой функции освобождаются все ресурсы, которые были созданы в функции `OnD3D10CreateDevice()`. Давайте, чтобы не запутаться, кратко рассмотрим все функции по порядку. Начнем с общей части, с тех функций, которые не зависят от версии Direct3D.

- `OnFrameMove()` вызывается перед прорисовкой кадра, в этой функции удобно осуществлять расчет нового положения объектов сцены (изменение мировой матрицы).
- `OnKeyboard()` осуществляет обработку нажатий клавиш на клавиатуре.
- `OnMouse()` обрабатывает нажатия на кнопки и вращение колесика мыши.
- `MsgProc()` служит для того, чтобы дать приложению возможность самостоятельно обработать поступившие сообщения, аналог оконной процедуры `WndProc()` из наших прошлых проектов.
- `ModifyDeviceSettings()` вызывается сразу после создания устройства Direct3D и позволяет при необходимости изменить параметры устройства.
- `OnDeviceRemoved()` вызывается в том случае, если устройство по каким-то причинам было удалено из системы (применительно к Direct3D 10: удаление устройства — событие достаточно редкое, если уж оно случилось, скорее всего, имеют место какие-то аппаратные проблемы с видеокартой или сбой драйвера, возможно также физическое отключение устройства).

Как можно заметить, предусмотрены решения для всех основных задач, при необходимости нам остается лишь внести изменения в соответствующие функции. Переходим к другой части, непосредственно связанной с Direct3D 10.

- `IsD3D10DeviceAcceptable()` позволяет проанализировать параметры устройства и в случае, если они не подходят для нашего приложения, предпринять какие-то действия. Чтобы отказаться от использования устройства, функция должна вернуть значение FALSE. Эта функция вызывается DXUT для всех комбинаций настроек устройства, при необходимости можно, например, исключить из использования все режимы с программной реализацией графического конвейера (REF).

- `OnD3D10CreateDevice()`, как мы уже знаем, создает объекты Direct3D 10. Чтобы быть точным, в этой функции надо создавать объекты, не связанные с вторичным буфером устройства (например, шрифтовые и спрайтовые объекты, буферы вершин и т. д.).
- `OnD3D10ResizedSwapChain()` служит для создания объектов, которые зависят от вторичного буфера устройства (например, элементы интерфейса, которые должны менять свое положение в зависимости от размеров окна программы).
- `OnD3D10FrameRender()` производит прорисовку сцены с помощью Direct3D 10, можно считать эту функцию расширенным аналогом функции `RenderScene()` из наших прежних проектов.
- `OnD3D10ReleasingSwapChain()` освобождает объекты, связанные с цепочкой переключений, которые были созданы в функции `OnD3D10ResizedSwapChain()`. Функция вызывается перед освобождением цепочки переключений, что случается при изменении размеров окна приложения, когда нужно изменить размеры вторичных буферов.
- `OnD3D10DestroyDevice()` освобождает объекты, которые были созданы в функции `OnD3D10CreateDevice()`. Функция вызывается каркасом DXUT перед освобождением устройства Direct3D 10.

После настройки функций обратного вызова идет инициализация DXUT. Она представляет собой последовательный вызов нескольких функций. Первой следует функция `DXUTInit()`. В зависимости от переданных параметров эта функция выполняет разбор командной строки, выводит сообщения об ошибках, а также использует дополнительные параметры командной строки. Прототип этой функции выглядит следующим образом:

```
HRESULT WINAPI DXUTInit(
 bool bParseCommandLine = true,
 bool bShowMsgBoxOnError = true,
 WCHAR* strExtraCommandLineParams = NULL,
 bool bThreadSafeDXUT = false
);
```

Вот что значат параметры функции:

- `bParseCommandLine` — флаг, который указывает, следует ли производить анализ параметров командной строки, по умолчанию разбор командной строки производится;
- `bShowMsgBoxOnError` — параметр, отвечающий за вывод сообщений об ошибках, которые по умолчанию выводятся;

- `strExtraCommandLineParams` — указатель на дополнительные параметры командной строки, который по умолчанию принимает значение `NULL`;
- `bThreadSafeDXUT` — флаг, определяющий, поддерживают ли объекты DXUT многопоточность, которая по умолчанию не поддерживается.

Следующая функция инициализации — `DXUTSetCursorSettings()`, с ее помощью задаются параметры курсора при работе в полноэкранном режиме. Рассмотрим прототип этой функции:

```
void WINAPI DXUTSetCursorSettings(
 bool bShowCursorWhenFullScreen = false,
 bool bClipCursorWhenFullScreen = false
);
```

Параметров здесь всего два, вот что они означают:

- `bShowCursorWhenFullScreen` — флаг видимости курсора, указывает, нужно ли отображать курсор в полноэкранном режиме. По умолчанию курсор в полноэкранном режиме не отображается.
- `bClipCursorWhenFullScreen` — флаг ограничения, он указывает, нужно ли ограничивать передвижение курсора границами экрана в полноэкранном режиме.

Далее следует функция для создания окна: `DXUTCreateWindow()`, по назначению похожа на функцию `InitWindow()`, которой мы с вами пользовались в предыдущих проектах. Внутрь мы заглядывать не будем, нас пока интересуют только параметры функции. Вот ее прототип:

```
HRESULT DXUTCreateWindow(
  CONST const WCHAR * strWindowTitle,
  HINSTANCE hInstance,
  HICON hIcon,
  HMENU hMenu,
  INT x,
  INT y
);
```

Теперь опишем ее параметры:

- `strWindowTitle` — указатель на строку, содержащую заголовок окна. Строка должна использовать символы Unicode (мы с этим уже не раз сталкивались, напомню, такая строка выглядит, например, так: `L"Заголовок окна"`). По умолчанию используется текст "Direct3D Window".
- `hInstance` — описатель экземпляра приложения либо значение `NULL` для получения описателя текущего модуля. Значение по умолчанию — `NULL`.

- `hIcon` — описатель значка приложения, либо значение `NULL`, если нужно использовать первый значок, включенный в исполняемый файл. По умолчанию здесь стоит `NULL`.
- `hMenu` — описатель ресурса меню либо значение `NULL`, чтобы указать, что меню отсутствует. Значение по умолчанию — `NULL`.
- `x` и `y` — горизонтальная и вертикальная координаты верхнего левого угла окна в системе координат экрана. Использование значения `CW_USEDEFAULT` позволяет Windows самой определить соответствующее положение. По умолчанию используется `CW_USEDEFAULT`.

Обратите внимание, что в нашем случае эта функция вызывается с минимальным набором параметров — текстом для заголовка окна, остальные значения оставлены по умолчанию.

Точно так же, как это делали мы в своих программах, за созданием окна приложения следует создание устройства Direct3D, осуществляемое здесь функцией `DXUTCreateDevice()`. Разберем ее прототип:

```
HRESULT DXUTCreateDevice(
  bool bWindowed,
  INT nSuggestedWidth,
  INT nSuggestedHeight
);
```

Параметры имеют следующие значения:

- `bWindowed` — указывает, в каком режиме запускается приложение, если передано значение `TRUE`, приложение запустится в оконном режиме, и в полноэкранном, если передано значение `FALSE`. По умолчанию установлено значение `TRUE`.
- `nSuggestedWidth` — требуемая начальная ширина вторичного буфера в пикселах. Указанное значение может быть скорректировано, принимая во внимание ограничения операционной системы и аппаратные возможности. Значение по умолчанию — 0.
- `nSuggestedHeight` — требуемая начальная высота вторичного буфера в пикселах. Указанное значение может быть скорректировано, принимая во внимание ограничения операционной системы и аппаратные возможности. Значение по умолчанию — 0. Если оба параметра `nSuggestedWidth` и `nSuggestedHeight` равны нулю, будут использованы соответствующие размеры клиентской области окна.

Здесь нужно обязательно отметить, что версия Direct3D, для которой создается устройство, в данном случае определяется каркасом автоматически. Решение принимается на основании того, поддержка какой версии Direct3D

реализована в системе и какие функции обратного вызова установлены. После создания устройства начинается основной цикл визуализации. Цикл "спрятан" внутри функции, но для нас с вами он пока и не важен: мы помним, что для прорисовки кадра вызываются функции `OnFrameMove()` и `OnD3D10FrameRender()`.

Итак, мы рассмотрели функцию `wWinMain()`, частично затронув общую логику работы программы с DXUT. Давайте коротко подведем итоги:

- Работа DXUT основана на функциях обратного вызова.
- Эти функции вызываются каркасом как реакция на некоторые события. Рутинные операции, такие как создание окна, создание устройства Direct3D 10, переключение в полноэкранный режим и обратно, а также некоторые другие, полностью выполняются каркасом DXUT.
- Нам остается только наполнить функции обратного вызова нужным нам содержанием в зависимости от результата, который мы хотим получить.

Примерная последовательность вызовов функций DXUT представлена на рис. 9.4.

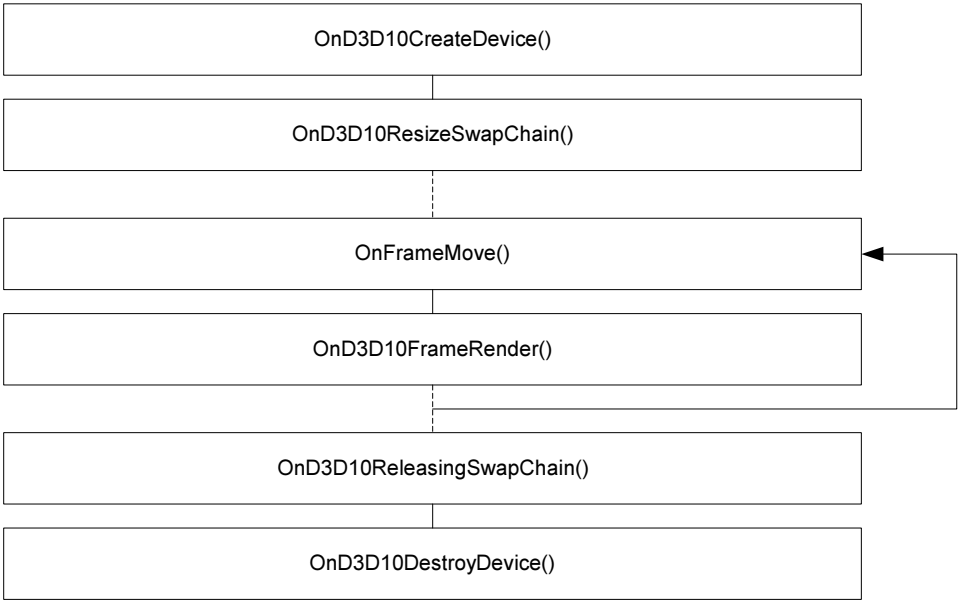

Рис. 9.4. Последовательность вызовов функций обратного вызова DXUT

Рисунок показывает, что перед началом основного цикла вызываются функция `OnD3D10CreateDevice()`, отвечающая за создание устройства, и функция

OnD3D10ResizedSwapChain(), отвечающая за изменение размеров буферов цепочки переключений. В самом цикле вызываются функции расчета нового положения объектов OnFrameMove() и прорисовки трехмерной сцены OnD3D10FrameRender(). Перед выходом из программы происходит подготовка к освобождению цепочки переключений: вызов функции OnD3D10ReleasingSwapChain(), после чего выполняются действия (например, очистка памяти), предшествующие уничтожению устройства Direct3D 10. Эта схема справедлива для случая, когда не происходит обработки других событий: нажатий клавиш на клавиатуре, изменения размеров окна приложения и т. д.

Вывод текста на экран

Давайте теперь, в качестве следующего шага, попробуем вывести текст на экран с помощью класса CDXUTTextHelper, который имеется в DXUT. Сделаем копию папки с пустым проектом DXUT, изменения будем вносить именно в нее. Для использования вспомогательного класса вывода текста нам нужно подключить заголовочный файл SDKmisc.h:

```
#include "SDKmisc.h"
```

Также необходимо добавить в проект файлы SDKmisc.h и SDKmisc.cpp. В папке проекта есть папка DXUT, все файлы в наши проекты мы будем добавлять из папки Optional, которая в ней находится.

Перед тем как приступить к редактированию, вспомним, как происходит вывод текста с помощью Direct3D. Фактически текст рисуется в спрайт, в картинку, которая затем переносится на экран. Здесь все происходит точно таким же образом. Объявим дополнительные глобальные переменные:

```
D3DX10Font*                 g_pFont = NULL;
ID3DX10Sprite*              g_pSprite = NULL;
CDXUTTextHelper*            g_pTxtHelper = NULL;
```

Переменная g_pFont имеет такое же назначение, что и в программе вывода текста из *гл. 4* — это указатель на шрифтовой объект. Далее идет указатель на спрайтовый объект, мы тоже с ним уже встречались. Третья переменная — указатель на экземпляр класса CDXUTTextHelper. Раньше мы создавали объекты в функции InitDirect3d10(), размещая их после создания устройства. Теперь, когда устройство Direct3D 10 за нас создает каркас DXUT, создание объектов располагается в функции OnD3D10CreateDevice(). Посмотрим, как это может выглядеть:

```
HRESULT CALLBACK OnD3D10CreateDevice(ID3D10Device* pd3dDevice,
    const DXGI_SURFACE_DESC* pBackBufferSurfaceDesc, void* pUserContext)
```

```
{
  HRESULT hr;
  V_RETURN( D3DX10CreateFont(pd3dDevice, 16, 0, FW_BOLD, 1, FALSE,
            DEFAULT_CHARSET, OUT_DEFAULT_PRECIS, DEFAULT_QUALITY,
            DEFAULT_PITCH | FF_DONTCARE, L"Arial", &g_pFont) );
  V_RETURN( D3DX10CreateSprite( pd3dDevice, 1, &g_pSprite ) );
  g_pTxtHelper = new CDXUTTextHelper(g_pFont, g_pSprite, 15 );
  return S_OK;
}
```

Обратим внимание на параметры, которые передаются в функцию каркасом:

- `pd3dDevice` — указатель на интерфейс устройства Direct3D 10, которое было создано;
- `pBackBufferSurfaceDesc` — указатель на структуру типа `DXGI_SURFACE_DESC`, которая содержит параметры вторичных буферов цепочки переключений;
- `pUserContext` — указатель на массив данных пользователя, в наших проектах мы его использовать не будем.

В теле функции мы видим создание шрифтового и спрайтового объектов с помощью уже знакомых нам функций `D3DX10CreateFont()` и `D3DX10CreateSprite()`, только они, почему-то, еще находятся внутри макроса `V_RETURN`. Тут следует сделать небольшое отступление.

В заголовочных файлах DXUT (точнее, в файле Dxstdafx.h) объявлены макросы, упрощающие контроль ошибок и составление программ. Нам больше не придется писать блоки из одинаковых строк, которые отличаются только именами переменных. Первый из таких макросов, с которым мы столкнулись, и есть `V_RETURN`. Что он делает? Посмотрим, как выглядит его описание:

```
#define V_RETURN(x)
{
  hr = x;
  if( FAILED(hr) )
  {
     return DXUTTrace( __FILE__, (DWORD)__LINE__, hr, L#x, TRUE );
  }
}
```

Ничего не напоминает? Да, именно так мы контролировали успешность завершения работы функций, когда писали программы в первой части книги. Для работы макроса должна быть объявлена переменная `hr` типа `HRESULT`. В эту переменную с помощью макроса помещается результат работы функ-

ции (она обозначена как "x"), если функция завершилась с ошибкой, код ошибки в переменной `hr` будет возвращен вызывающей функции и будет выведено окно с указанием имени файла, содержащего исходный текст, строки, в которой ошибка возникла, а также ее код. При этом сообщение об ошибке еще будет продублировано и в сообщениях отладчика Visual Studio. Кроме макроса `V_RETURN` еще есть макрос `V`, он отличается только тем, что не возвращает значение переменной `hr` в вызывающую функцию. Существуют макросы для выполнения других полезных задач. С некоторыми из них мы познакомимся по ходу нашего путешествия.

Как создаются шрифтовой и спрайтовый объект и как контролируется результат выполнения функций, теперь ясно. Рассмотрим, как создается экземпляр класса `CDXUTTextHelper`. Мы используем конструктор класса, который имеет следующие параметры:

```
CDXUTTextHelper(
 ID3DX10Font* pFont10 = NULL,
 ID3DX10Sprite* pSprite10 = NULL,
 int nLineHeight = 15
);
```

Назначение параметров такое:

- `pFont10` — указывает, каким шрифтом выводить текст.
- `pSprite10` — указывает спрайтовый объект, с помощью которого осуществляется вывод текста.
- `nLineHeight` — задает высоту строки текста в пикселах, другими словами, на сколько пикселов вниз смещается позиция вывода текста при переходе на следующую строку.

> ***Замечание***
>
> Класс `CDXUTTextHelper` может одновременно использоваться как устройством Direct3D 10, так и устройством Direct3D 9, для чего имеется соответствующий вариант конструктора.

Объекты для вывода текста мы создали. Давайте сразу же обеспечим их корректное освобождение при выходе из программы. Открываем функцию `OnD3D10DestroyDevice()`, освобождение объектов будем осуществлять таким образом:

```
void CALLBACK OnD3D10DestroyDevice( void* pUserContext )
{
  SAFE_DELETE( g_pTxtHelper );
  SAFE_RELEASE( g_pSprite );
```

```
  SAFE_RELEASE( g_pFont );
}
```

Здесь мы также используем макросы DXUT. Удаление экземпляра класса `CDXUTTextHelper` с помощью макроса `SAFE_DELETE` в результате происходит таким же образом, как в следующем фрагменте программы:

```
if (g_pTxtHelper) {
  delete(g_pTxtHelper);
  g_pTxtHelper = NULL;
}
```

Для освобождения спрайтового объекта, если бы мы не использовали макрос `SAFE_RELEASE`, нам бы пришлось написать такую строку:

```
if (g_pSprite) g_pSprite->Release();
```

С макросом все намного проще и текст программы стал более читаемым. За корректное освобождение памяти мы теперь спокойны, и можно сосредоточиться непосредственно на выводе текста. Как мы знаем, рисование кадра происходит в функции `OnD3D10FrameRender()`, туда мы и отправимся. Представим, что нам нужно вывести на экран надпись "Привет!!!", в этом случае функция прорисовки кадра будет содержать такой текст:

```
void CALLBACK OnD3D10FrameRender( ID3D10Device* pd3dDevice,
                  double fTime, float fElapsedTime, void* pUserContext )
{
  // Очистить буфер визуализации и шаблонный буфер глубины
  float ClearColor[4] = { 0.176f, 0.196f, 0.667f, 0.0f };
  ID3D10RenderTargetView* pRTV = DXUTGetD3D10RenderTargetView();
  pd3dDevice->ClearRenderTargetView( pRTV, ClearColor );

  ID3D10DepthStencilView* pDSV = DXUTGetD3D10DepthStencilView();
  pd3dDevice->ClearDepthStencilView( pDSV, D3D10_CLEAR_DEPTH, 1.0, 0 );

  g_pTxtHelper->Begin();
  g_pTxtHelper->SetInsertionPos( 5, 5 );
  g_pTxtHelper->SetForegroundColor( D3DXCOLOR( 1.0f, 1.0f, 1.0f, 1.0f ));
  g_pTxtHelper->DrawTextLine( L"Привет!!!" );
  g_pTxtHelper->End();
}
```

Рассмотрим параметры, которые DXUT передает функции рисования кадра:

- `pd3dDevice` — указатель на устройство Direct3D 10;
- `fTime` — время, прошедшее с момента запуска программы, в секундах;

- `fElapsedTime` — время, прошедшее с предыдущего вызова функции визуализации (временной промежуток между двумя кадрами), в секундах;
- `pUserContext` — указатель на массив данных пользователя, его мы в наших проектах не используем.

Теперь мы знаем, какие данные передаются в функцию. Разберем тело функции построчно. В первой строке задается цвет, которым будет заполнен буфер визуализации. Это делается так же, как мы всегда делали, разница только в самом цвете:

```
float ClearColor[4] = { 0.176f, 0.196f, 0.667f, 0.0f };
```

Далее мы должны заполнить буфер визуализации указанным цветом. Для этого нам нужен указатель на соответствующее представление данных. Когда мы сами создавали устройство, мы сами все контролировали и все указатели были у нас под рукой. При использовании DXUT нам придется запрашивать этот указатель у каркаса с помощью функции `DXUTGetD3D10RenderTargetView()`. Эта функция не требует параметров и возвращает нужный нам указатель:

```
ID3D10RenderTargetView* pRTV = DXUTGetD3D10RenderTargetView();
```

Указатель на представление данных теперь известен, очищаем буфер визуализации привычным способом:

```
pd3dDevice->ClearRenderTargetView( pRTV, ClearColor );
```

После этого должна идти очистка шаблонного буфера глубины, она выполняется точно таким же образом, только для получения указателя на представление данных используется функция `DXUTGetD3D10DepthStencilView()`:

```
ID3D10DepthStencilView* pDSV = DXUTGetD3D10DepthStencilView();
pd3dDevice->ClearDepthStencilView( pDSV, D3D10_CLEAR_DEPTH,1.0, 0 );
```

На этом вспомогательные операции закончены, приступаем к выводу текста. Первым всегда вызывается метод `CDXUTTextHelper::Begin`, он сигнализирует о начале вывода данных, относящихся к тексту:

```
g_pTxtHelper->Begin();
```

Устанавливаем координаты точки вставки текста, они представляют собой координаты левого верхнего угла первого символа в строке:

```
g_pTxtHelper->SetInsertionPos( 5, 5 );
```

Зададим цвет, которым будет выводиться текст, пусть у нас это будет белый цвет:

```
g_pTxtHelper->SetForegroundColor( D3DXCOLOR( 1.0f, 1.0f, 1.0f, 1.0f ) );
```

В следующей строке происходит вывод строки текста согласно установленным параметрам, мы используем метод `CDXUTTextHelper::DrawTextLine`:

```
g_pTxtHelper->DrawTextLine( L"Привет!!!" );
```

Так как мы уже вывели весь текст, нужно вызвать метод `CDXUTTextHelper::End`, тем самым, указывая на окончание работы с текстом:

```
g_pTxtHelper->End();
```

Вот и все, можно компилировать и запускать проект. Результат наших усилий должен быть похож на изображение, показанное на рис. 9.5.

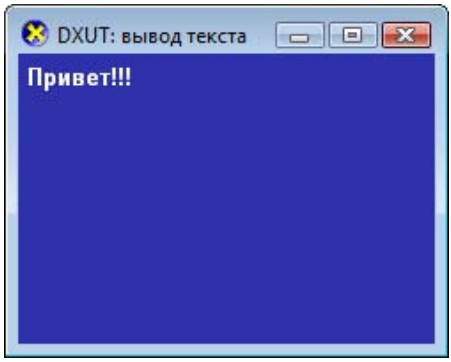

Рис. 9.5. Вывод текста с помощью класса DXUT

Может показаться, что здесь вывод текста организован сложнее. Может быть, отчасти так оно и есть, но с помощью вспомогательного класса мы приобретаем много дополнительных возможностей. Скопируем строку, в которой выводится текст еще раз, чтобы получилось такая последовательность:

```
g_pTxtHelper->DrawTextLine( L"Привет!!!" );
g_pTxtHelper->DrawTextLine( L"Привет!!!" );
```

Запустим программу. Мы автоматически получили две строки текста, идущие подряд (рис. 9.6).

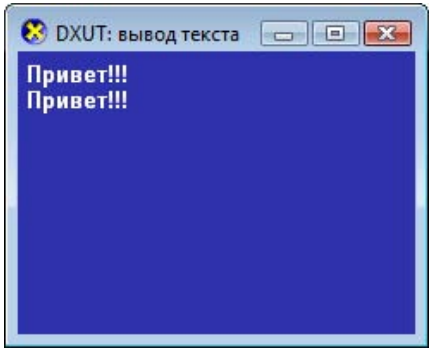

Рис. 9.6. Вывод нескольких строк текста

Этим возможности DXUT, конечно, не ограничиваются. Давайте посмотрим, как вывести на экран информацию о режиме вывода графики и количестве кадров в секунду. Воспользуемся функциями `DXUTGetFrameStats()` и `DXUTGetDeviceStats()`, вставим их вместо текста в имеющиеся две строки (параметр `TRUE` означает, что в выводимую информацию нужно включить количество кадров в секунду):

```
g_pTxtHelper->DrawTextLine( DXUTGetFrameStats( TRUE ) );
g_pTxtHelper->DrawTextLine( DXUTGetDeviceStats() );
```

Компилируем, запускаем проект. В окне программы отображаются две строки с системной информацией (рис. 9.7).

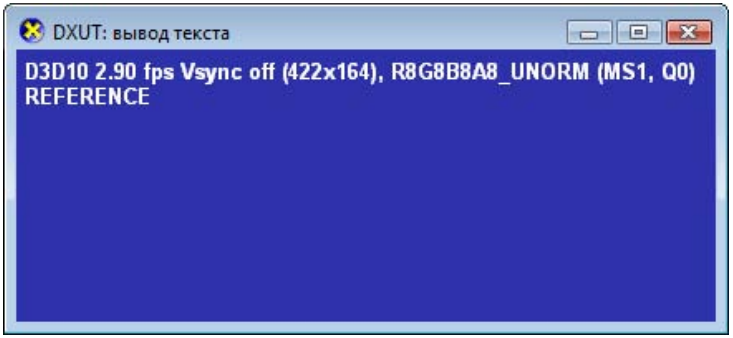

Рис. 9.7. Вывод системной информации

Первая строка сообщает нам, какая версия Direct3D используется (D3D10), количество кадров в секунду (2,9), формат вторичного буфера (R8G8B8A8_UNORM) и некоторые другие сведения. Во второй строке отображается тип устройства Direct3D (REFERENCE).

Возможно, эти данные будут нам полезны, но как быть, если мы хотим вывести какие-то свои значения, например, нам нужно отобразить только количество кадров в секунду? Количество кадров в секунду возвращает функция `DXUTGetFPS()`, но вывести его с помощью метода `CDXUTTextHelper::DrawTextLine` у нас не получится, так как этот метод рассчитан исключительно на вывод текста, а функция возвращает число типа `float`. Для вывода текста и значений переменных служит метод `CDXUTTextHelper::DrawFormattedText`. Отредактируем строки вывода текста следующим образом:

```
g_pTxtHelper->
        DrawFormattedTextLine( L"Количество кадров в секунду: %.2f",
                               DXUTGetFPS() );
```

```
g_pTxtHelper->SetForegroundColor( D3DXCOLOR( 0.0f, 1.0f, 0.0f, 1.0f ) );
g_pTxtHelper->DrawTextLine( L"Нажмите <ESC> для выхода" );
```

Вот что получится, если запустить программу (рис. 9.8): в первой строке белым цветом будет отображаться количество кадров в секунду, а в следующей строке — сообщение о нажатии клавиши <ESC>, но уже зеленым цветом.

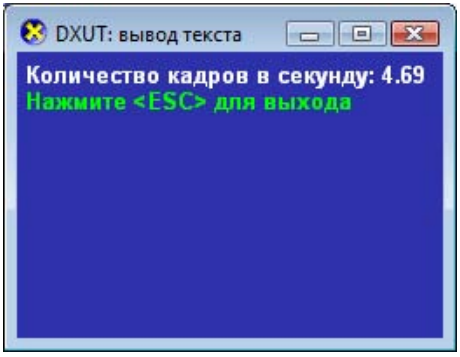

Рис. 9.8. Вывод количества кадров в секунду

Если вы когда-нибудь пользовались функцией `printf()`, параметры метода `DrawFormattedText` покажутся вам знакомыми. На всякий случай коротко поясню. Мы выводим текст как обычно, и в нем вставляем специальные "маркеры" (управляющие последовательности), которые указывают взять значение переменной из списка переменных после строки текста. "Маркер" всегда начинается с символа "%" ("процент"). С помощью "маркера" указывается тип переменной и, если необходимо, параметры вывода значения. Например, в нашем случае, указывается тип значения (f — двойной точности) и количество знаков после десятичной точки (.2 — два знака).

> ***Замечание***
>
> При необходимости включить в текст сам символ "%" его необходимо удвоить: %%. Например, для вывода текста "Работа завершена на 100%" в программе нужно использовать следующую строку: `L"Работа завершена на 100%%"`.

Если в тексте есть несколько "маркеров", для первого будет использовано значение первой переменной в списке, для второго — значение второй переменной и т. д. Список переменных записывается через запятую. В табл. 9.1

приведены некоторые типы значений выводимых переменных. С полным списком можно ознакомиться в документации к языку С.

Таблица 9.1. *Типы выводимых значений переменных для метода DrawFormattedText*

Тип	Описание
%d %i	Десятичное целое число со знаком
%f	Число двойной точности. Количество цифр перед десятичной точкой зависит от величины числа, количество цифр после запятой зависит от заданной точности
%s	Строка с завершающим нулем
%x	Шестнадцатеричное число без знака, для вывода используются символы "abcdef"
%X	Шестнадцатеричное число без знака, для вывода используются символы "ABCDEF"

Чтобы лучше понять смысл параметров, попробуйте ввести следующие строки, запустите программу и посмотрите, что получится.

```
g_pTxtHelper->
        DrawFormattedTextLine( L"Количество кадров в секунду: %.2f",
                               DXUTGetFPS() );
g_pTxtHelper->
        DrawFormattedTextLine( L"Число с заданной точностью: %.3f",
                               1.23456f );
g_pTxtHelper->
        DrawFormattedTextLine( L"Шестнадцатеричное число: %X", 6912);
g_pTxtHelper->
        DrawFormattedTextLine( L"Строка: %s", L"Это строка текста");
g_pTxtHelper->SetForegroundColor( D3DXCOLOR( 0.0f, 1.0f, 0.0f, 1.0f ) );
g_pTxtHelper->DrawTextLine( L"Нажмите <ESC> для выхода" );
```

Результат можно увидеть на рис. 9.9.

С выводом текста, наверное, теперь все ясно. Проект, в котором выводятся на экран количество кадров в секунду и сообщение о нажатии клавиши <ESC>, можно найти на прилагаемом компакт-диске в директории Glava9\ DXUT_Text.

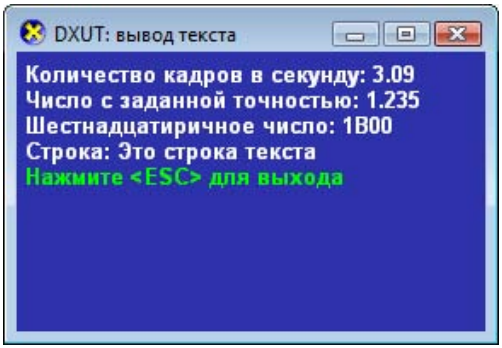

Рис. 9.9. Использование различных типов при выводе значений в тексте

Мы с вами научились выводить многострочный текст и, где требуется, вставлять значения переменных. Это, безусловно, большой шаг вперед. Продолжим свое знакомство с DXUT, ведь умение выводить текст — это хорошо, но все же главное для нас — это вывод трехмерной графики.

Глава 10

Загружаем объемные модели

До сих пор нам приходилось вводить координаты трехмерных объектов вручную. Поступать таким образом, вероятно, не самая блестящая идея. Если ввод координат вершин треугольника особых усилий не требует, то чтобы ввести координаты для вершин куба, уже придется постараться, особенно если для каждой вершины нужно задать координаты в пространстве, нормаль, цвет и текстурные координаты. Можно представить себе, сколько времени уйдет, чтобы вручную ввести координаты для модели машины или самолета, которые нам вдруг потребовалось бы использовать, и с какого раза получилось бы правильно задать все эти координаты. Как вы понимаете, так трехмерные модели не делаются. Существует множество редакторов, в которых можно с помощью удобных средств редактирования смоделировать нужный объект, сохранить в файл, а потом в своей программе прочитать данные из этого файла и использовать модель в трехмерной сцене. В каком формате сохранять файл для использования в программе? Вот здесь начинаются трудности. В предыдущих версиях Direct3D, по девятую включительно, поддерживался формат X File (.x). В файлах этого формата можно было хранить информацию самого разного рода, в том числе и трехмерные модели. В Direct3D 10 поддержка для формата .x отсутствует и, похоже, вряд ли когда-то появится. Вместо него для хранения моделей используется формат файлов моделей DXUT (.sdkmesh). В документации подчеркивается, что этот формат предназначен исключительно для примеров, иллюстрирующих работу с DirectX SDK. Более серьезные программы потребуют разработки собственного формата для хранения моделей объектов, но для учебных целей формат .sdkmesh нам вполне подойдет.

Как создать файл формата .sdkmesh?

Если с созданием файлов формата X File ситуация более или менее понятна, то с новым форматом есть некоторые проблемы. Для формата .x имеется множество различных утилит и модулей экспорта, которые позволяют преобразовывать и экспортировать модели из программ трехмерного моделирования. Для формата .sdkmesh пока еще нет ничего, кроме программы meshconvert, которая поставляется вместе с DirectX SDK. Эта программа может преобразовать файл формата .x в файл формата .sdkmesh, и это, видимо, пока единственный общедоступный способ получения файлов такого формата. Таким образом, нам требуется выполнить следующие действия.

Создаем модель объекта, который мы хотим использовать в нашей программе. Каким-то способом экспортируем готовую модель в формат .x.

Используем программу meshconvert для преобразования в формат .sdkmesh.

Рассмотрим каждый пункт подробнее. В качестве примера возьмем программу 3DS MAX 8 и бесплатный модуль для экспорта в формат .x от Panda Software, доступный по адресу **http://www.andytather.co.uk/Panda/directxmax.aspx**.

Экспорт в формат .x

Загружаем программу 3DS MAX 8, можно использовать и любую другую версию, лишь бы для нее имелся соответствующий модуль экспорта. Создаем нужную модель или берем заранее приготовленную, накладываем текстуры, если это еще не сделано. В общем, модель должна иметь такой вид, в каком она должна появиться в программе. В качестве примера я подготовил куб с наложенной текстурой (рис. 10.1).

Посмотрим, как его экспортировать в формат .x. Перед экспортированием нужный объект необходимо выделить, после чего в главном меню 3DS MAX 8 выбираем **File | Export…** (Файл | Экспорт…). Нужно указать тип файла "Panda DirectX (.X)" в выпадающем списке, открыть папку, куда нужно сохранить файл (например, C:\Mesh), ввести имя файла ("box.x") и щелкнуть кнопку **Save** (Сохранить).

Появится окно с параметрами преобразования (рис. 10.2), можно оставить все как есть, важно лишь убедиться, что установлен экспорт в левостороннюю систему координат: на вкладке **X File Settings** (Настройки формата X File) должен быть установлен флажок **Left Handed Axis** (Левосторонняя система координат), как показано на рис. 10.3.

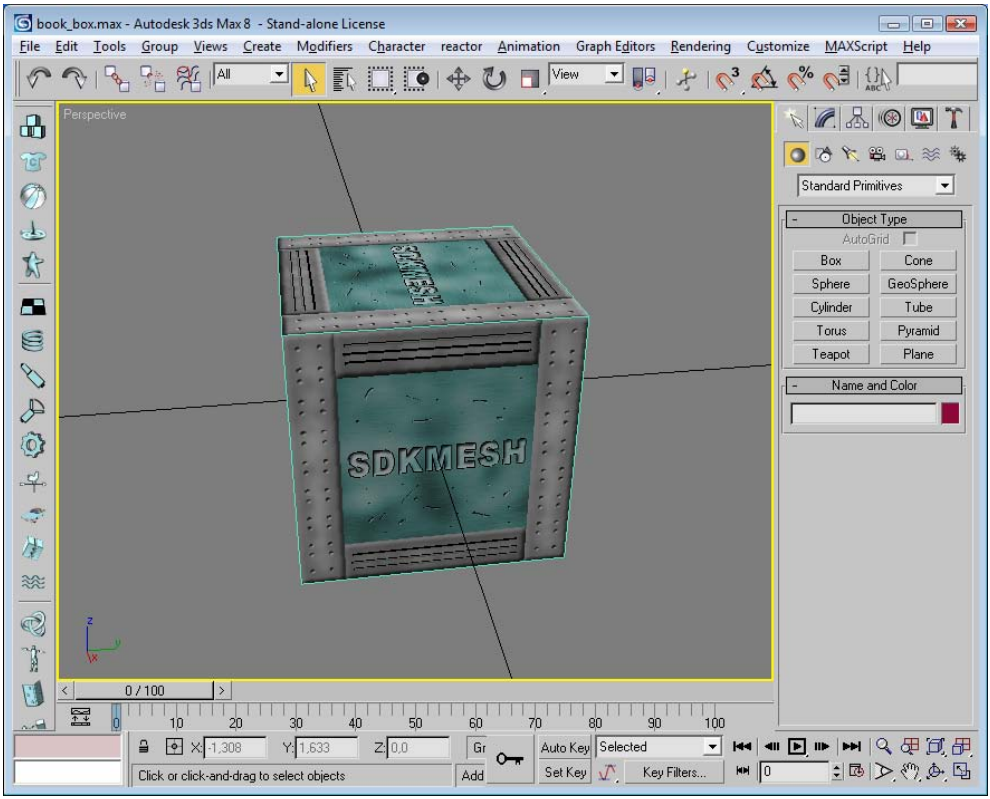

Рис. 10.1. Модель для экспорта в формат .x

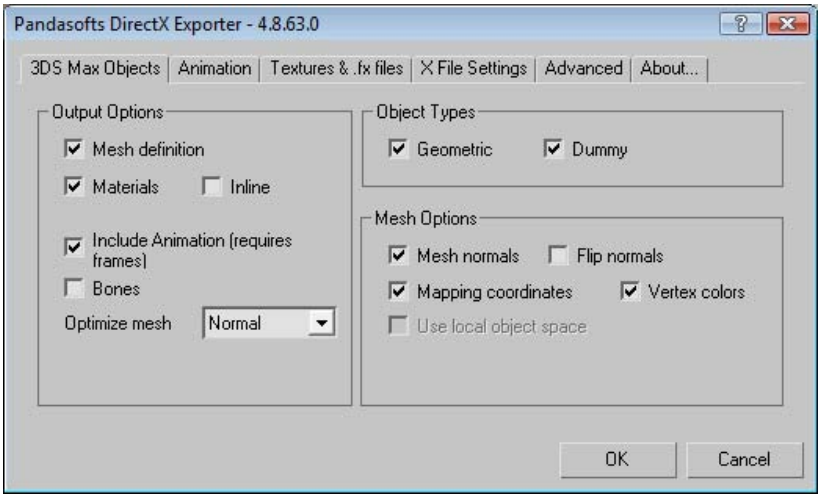

Рис. 10.2. Окно параметров экспорта

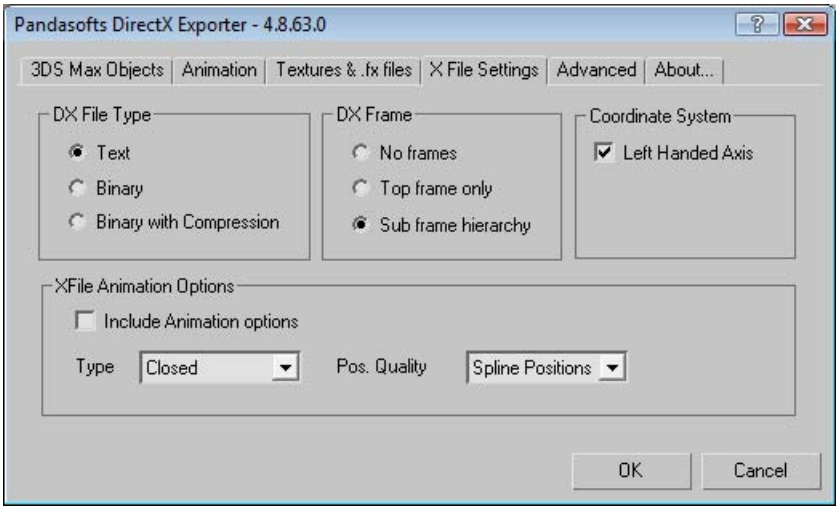

Рис. 10.3. Настройка координатной системы для экспорта

Если флажок не установлен, мы получим на модели зеркально отраженные текстуры, что может доставить много неприятных минут и отнять время на поиск несуществующей ошибки в программе.

Программа meshconvert

Если мы все сделали правильно, на диске должен появиться файл с расширением x. Преобразуем его в файл .sdkmesh. Рассмотрим программу meshconvert, которая вызывается так:

meshconvert [опции] [имя входного файла]

Вот что означают возможные параметры командной строки:

- имя входного файла — представляет имя файла, содержащего модель, которую нужно преобразовать. Входной файл может иметь формат .x, .obj или .sdkmesh.
- Опции могут выражаться следующими ключами:
  - /o [файл] — с помощью этого ключа указывается выходной файл, который будет содержать результат преобразования;
  - /y — этот ключ указывает на необходимость переписать имеющийся выходной файл;
  - /sdkmesh — значение по умолчанию, запрашивающее преобразование в файл формата .sdkmesh;

- /x — запрос преобразования в двоичный формат .x;
- /xt — запрос преобразования в текстовый формат .x.

Чтобы преобразовать файл box.x в файл box.sdkmesh, нам придется выполнить следующие операции. В первую очередь нужно запустить программу командного процессора: **Пуск | Выполнить**…, в появившемся поле ввода набрать "cmd" и нажать клавишу <Enter>. Перейти в папку, где содержится исходный файл .x с моделью, например: "cd C:\Mesh". Для запуска преобразования введем вот такую команду: "meshconvert /o box.sdkmesh box.x". Все команды, как вы понимаете, вводятся без кавычек. В результате в той же папке, рядом с исходным файлом, у нас появится файл в формате .sdkmesh, который мы уже без каких-либо изменений сможем использовать в своей программе. Сейчас мы напишем приложение, которое сможет загрузить модель из файла и вывести ее на экран.

Вывод модели на экран

Что нам понадобится для того, чтобы загрузить модель из файла? Инициализация программы будет во многом похожа на инициализацию, применявшуюся для вывода треугольника или куба. Вот действия, которые предстоит выполнить:

1. Создать эффект из файла.
2. Извлечь технику отображения.
3. Настроить связь с переменными шейдеров.
4. Описать формат входных данных.
5. Создать объект входных данных и связать его с графическим конвейером.
6. Загрузить модель из файла с помощью метода `CDXUTSDKMesh::Create`.
7. Установить соответствующим образом матрицы.
8. Добавить в функцию визуализации вызов метода `CDXUTSDKMesh::Render`.
9. Обеспечить корректное освобождение памяти при выходе из программы.

Пункты с первого по пятый включительно мы с вами проделывали уже не один раз, их мы подробно рассматривать не будем, а вот на остальных остановимся поподробнее. Сделаем копию папки с проектом, в котором мы выводили на экран текст, он будет нашей отправной точкой. В первую очередь мы должны добавить в программу подключение заголовочного файла SDKMesh.h, в проект также нужно добавить файлы SDKMesh.h

и SDKmesh.cpp. Новые глобальные переменные для вывода модели с наложенной текстурой:

```
ID3D10Effect*                g_pEffect = NULL;
ID3D10InputLayout*           g_pVertexLayout = NULL;
ID3D10EffectTechnique*       g_pTechnique = NULL;
ID3D10EffectMatrixVariable* g_pWorldVariable = NULL;
ID3D10EffectMatrixVariable* g_pViewVariable = NULL;
ID3D10EffectMatrixVariable* g_pProjectionVariable = NULL;
ID3D10EffectShaderResourceVariable* g_pTexResource = NULL;

CDXUTSDKMesh                 g_Mesh;

D3DXMATRIX                   g_World;
D3DXMATRIX                   g_View;
D3DXMATRIX                   g_Projection;
```

Здесь мы практически со всем знакомы, новым для нас является только объявление экземпляра класса `CDXUTSDKMesh`. Пока мы на время забудем о нем, но скоро с ним встретимся. А сейчас сразу отправимся в функцию `OnD3D10CreateDevice()`, где будет осуществляться инициализация. Новые строки будем вставлять перед оператором `return`. Вот как выглядит реализация первых трех пунктов нашего алгоритма (вы можете предложить свой вариант):

```
// Создаем эффект из файла
V_RETURN( D3DX10CreateEffectFromFile(L"Mesh.fx", NULL, NULL,
"fx_4_0", D3D10_SHADER_ENABLE_STRICTNESS, 0, pd3dDevice, NULL, NULL,
&g_pEffect, NULL, NULL) );

// Извлекаем технику отображения
g_pTechnique = g_pEffect->GetTechniqueByName( "RenderMesh" );

// Связь с переменными шейдеров
g_pTexResource = g_pEffect->
                 GetVariableByName( "g_txDiffuse" )->AsShaderResource();
g_pWorldVariable = g_pEffect->GetVariableByName( "World" )->AsMatrix();
g_pViewVariable = g_pEffect->GetVariableByName( "View" )->AsMatrix();
g_pProjectionVariable = g_pEffect->
                        GetVariableByName( "Projection" )->AsMatrix();
```

Следующим пунктом идет описание формата входных данных. Как вы думаете, какие данные модели нужно хранить в файле, чтобы иметь возможность накладывать на модель текстуру и выводить ее на экран с учетом освещения? Само собой, обязательно должны присутствовать координаты вершин в пространстве. Для наложения текстуры потребуются текстурные координаты, а для расчета освещения нужны нормали к каждой вершине. Именно эти данные и хранятся в файле .sdkmesh. То есть массив, описывающий формат данных, будет сдержать три элемента: координаты вершин, координаты нормалей и текстурные координаты:

```
// Описываем формат входных данных
const D3D10_INPUT_ELEMENT_DESC layout[] =
{
  { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
    D3D10_INPUT_PER_VERTEX_DATA, 0 },
  { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
    D3D10_INPUT_PER_VERTEX_DATA, 0 },
  { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24,
    D3D10_INPUT_PER_VERTEX_DATA, 0 }
};
UINT numElements = sizeof(layout)/sizeof(layout[0]);
```

После описания формата снова встречаем знакомые строки:

```
// Создаем объект входных данных
D3D10_PASS_DESC PassDesc;
g_pTechnique->GetPassByIndex( 0 )->GetDesc( &PassDesc );
V_RETURN( pd3dDevice->CreateInputLayout( layout, numElements,
          PassDesc.pIAInputSignature, PassDesc.IAInputSignatureSize,
          &g_pVertexLayout ) );

// Связываем объект входных данных с графическим конвейером
pd3dDevice->IASetInputLayout( g_pVertexLayout );
```

Сейчас наступает торжественный момент, мы загружаем трехмерную модель из файла, вызывая метод `CDXUTSDKMesh::Create`:

```
V_RETURN( g_Mesh.Create( pd3dDevice, L"box.sdkmesh", true ) );
```

Да, это все! Все в одной строчке. Помните, сколько у нас занимали координаты вершин в предыдущих проектах? Класс `CDXUTSDKMesh` сам выделяет память для загрузки модели, читает данные из файла, а также самостоятельно загрузит указанную в файле текстуру. Посмотрим, какие для этого требуются параметры. Первый параметр — указатель на устройство Direct3D 10, с кото-

рым будет связна модель, второй параметр, очевидно, — это имя файла формата .sdkmesh, содержащего данные модели. Третий параметр показывает, следует ли проводить оптимизацию данных при создании модели в памяти. Имеется еще несколько параметров, но их значения мы оставили по умолчанию. Для контроля над результатом выполнения функции используется макрос `V_RETURN`.

Инициализируем мировую матрицу и заполним матрицу вида:

```
// Инициализируем мировую матрицу
D3DXMatrixIdentity( &g_World );

// Инициализируем матрицу вида
D3DXVECTOR3 Eye( 0.0f, 2.0f, -3.0f );
D3DXVECTOR3 At( 0.0f, 0.0f, 0.0f );
D3DXVECTOR3 Up( 0.0f, 1.0f, 0.0f );
D3DXMatrixLookAtLH( &g_View, &Eye, &At, &Up );
```

Поскольку положение камеры у нас не изменяется, можно сразу передать матрицу вида в шейдер:

```
// Установим значение переменной шейдера для матрицы вида
g_pViewVariable->SetMatrix( (float*)&g_View );
```

Инициализация окончена, выходим из функции:

```
return S_OK;
```

После редактирования функция `OnD3D10CreateDevice()` должна принять следующий вид:

```
HRESULT CALLBACK OnD3D10CreateDevice( ID3D10Device* pd3dDevice,
                    const DXGI_SURFACE_DESC* pBackBufferSurfaceDesc,
                    void* pUserContext )
{
  HRESULT hr;
  // Создаем шрифтовой объект
  V_RETURN(D3DX10CreateFont(pd3dDevice, 16, 0, FW_BOLD, 1, FALSE,
           DEFAULT_CHARSET, OUT_DEFAULT_PRECIS, DEFAULT_QUALITY,
           DEFAULT_PITCH | FF_DONTCARE, L"Arial", &g_pFont) );

  // Создаем спрайтовый объект для вывода текста
  V_RETURN( D3DX10CreateSprite( pd3dDevice, 1, &g_pSprite ) );

  // Создаем экземпляр вспомогательного класса
  g_pTxtHelper = new CDXUTTextHelper( g_pFont, g_pSprite, 15 );
```

```
// Создаем эффект из файла
V_RETURN( D3DX10CreateEffectFromFile(L"Mesh.fx", NULL, NULL,
          "fx_4_0", D3D10_SHADER_ENABLE_STRICTNESS, 0, pd3dDevice,
          NULL, NULL, &g_pEffect, NULL, NULL) );

  // Извлекаем технику отображения
g_pTechnique = g_pEffect->GetTechniqueByName( "RenderMesh" );

// Связь с переменными шейдеров
g_pTexResource = g_pEffect->
                 GetVariableByName( "g_txDiffuse" )->AsShaderResource();
g_pWorldVariable = g_pEffect->GetVariableByName( "World" )->AsMatrix();
g_pViewVariable = g_pEffect->GetVariableByName( "View" )->AsMatrix();
g_pProjectionVariable = g_pEffect->
                 GetVariableByName( "Projection" )->AsMatrix();

// Описываем формат входных данных
const D3D10_INPUT_ELEMENT_DESC layout[] =
{
  { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
    D3D10_INPUT_PER_VERTEX_DATA, 0 },
  { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
    D3D10_INPUT_PER_VERTEX_DATA, 0 },
  { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24,
    D3D10_INPUT_PER_VERTEX_DATA, 0 },
};
UINT numElements = sizeof(layout)/sizeof(layout[0]);

// Создаем объект входных данных
D3D10_PASS_DESC PassDesc;
g_pTechnique->GetPassByIndex( 0 )->GetDesc( &PassDesc );
V_RETURN( pd3dDevice->CreateInputLayout( layout, numElements,
             PassDesc.pIAInputSignature, PassDesc.IAInputSignatureSize,
             &g_pVertexLayout ) );

// Связываем объект входных данных с графическим конвейером
pd3dDevice->IASetInputLayout( g_pVertexLayout );

// Загружаем модель
V_RETURN( g_Mesh.Create( pd3dDevice, L"box.sdkmesh", true ) );
```

```
  // Инициализируем мировую матрицу
  D3DXMatrixIdentity( &g_World );

  // Инициализируем матрицу вида
  D3DXVECTOR3 Eye( 0.0f, 2.0f, -3.0f );
  D3DXVECTOR3 At( 0.0f, 0.0f, 0.0f );
  D3DXVECTOR3 Up( 0.0f, 1.0f, 0.0f );
  D3DXMatrixLookAtLH( &g_View, &Eye, &At, &Up );

  // Установим значение переменной шейдера
  // для матрицы вида
  g_pViewVariable->SetMatrix( (float*)&g_View );

  return S_OK;
}
```

Мы уже сказали, что матрица вида у нас остается неизменной, а как обстоят дела с другими матрицами? Если мы будем, например, вращать модель, мировая матрица будет изменяться с каждым кадром, значит, нам нужно передавать ее в шейдер в функции `OnFrameMove()` или `OnD3D10FrameRender()`. Матрица проекции зависит от размеров окна, точнее, от соотношения его ширины и высоты, при изменении этих параметров матрица должна быть пересчитана. Раньше мы это делали в обработчике сообщения `WM_SIZE`, здесь нам пригодится функция `OnD3D10ResizedSwapChain()`, которая вызывается после изменения размеров окна. Вот как будет выглядеть установка матрицы проекции:

```
HRESULT CALLBACK OnD3D10ResizedSwapChain( ID3D10Device* pd3dDevice,
                          IDXGISwapChain *pSwapChain,
                          const DXGI_SURFACE_DESC* pBackBufferSurfaceDesc,
                          void* pUserContext )
{
  // Устанавливаем параметры матрицы проекции
  float fAspect = (float)( pBackBufferSurfaceDesc->Width )/
                              (float)( pBackBufferSurfaceDesc->Height );
  D3DXMatrixPerspectiveFovLH( &g_Projection, D3DX_PI*0.25f, fAspect,
                                0.5f, 100.0f );
  g_pProjectionVariable->SetMatrix( (float*)&g_Projection );

  return S_OK;
}
```

Какие параметры передаются в эту функцию и какие из них мы сейчас можем использовать? Познакомимся с прототипом функции `OnD3D10ResizedSwapChain()`, он представлен ниже:

```
HRESULT CALLBACK OnD3D10ResizedSwapChain(
  ID3D10Device* pd3dDevice,
  IDXGISwapChain *pSwapChain,
  const DXGI_SURFACE_DESC *pBackBufferSurfaceDesc,
  void* pUserContext
)
```

Параметры этой функции в основном повторяют параметры функции `OnD3D10CreateDevice()`:

- `pd3dDevice` — указатель на интерфейс устройства Direct3D 10;
- `pSwapChain` — указатель на интерфейс цепочки переключений;
- `pBackBufferSurfaceDesc` — указатель на структуру типа `DXGI_SURFACE_DESC`, которая содержит параметры вторичных буферов цепочки переключений, в том числе и их размеры;
- `pUserContext` — указатель на массив данных пользователя, в наших проектах мы его не используем.

В первую очередь, вычисляем соотношение сторон окна, используя данные структуры из `pBackBufferSurfaceDesc`, предварительно преобразовав значения ширины и высоты к типу `float`:

```
float fAspect = (float)( pBackBufferSurfaceDesc->Width )/
                         (float)( pBackBufferSurfaceDesc->Height );
```

После этого формируем матрицу проекции и передаем ее в соответствующую переменную шейдера:

```
D3DXMatrixPerspectiveFovLH( &g_Projection, D3DX_PI*0.25f, fAspect,
                            0.5f, 100.0f );
g_pProjectionVariable->SetMatrix( (float*)&g_Projection );
```

Матрица проекции установлена и будет пересчитываться при изменении размеров окна, а также при переключении программы в полноэкранный режим. Незаполненной у нас пока остается только мировая матрица. Давайте воспользуемся имеющимся опытом и будем вращать нашу модель вокруг оси Y, задавая угол вращения, исходя из использующегося драйвера устройства. Узнать, какой тип драйвера используется в данный момент, мы можем при помощи функции `DXUTGetDeviceSettings()`. Она возвращает структуру типа `DXUTDeviceSettings`, в которой имеется поле, содержащее тип драйвера.

Расчет мировой матрицы разместим внутри функции `OnFrameMove()`, так как для этого она подходит лучше. Вот как все это выглядит в программе:

```
void CALLBACK OnFrameMove( double fTime, float fElapsedTime,
                           void* pUserContext )
{
  // Получаем параметры устройства Direct3D
  DXUTDeviceSettings D3D10DevSet;
  D3D10DevSet = DXUTGetDeviceSettings();
  static float t = 0.0f;

  // Устанавливаем угол поворота в зависимости от
  // типа драйвера устройства
  if (D3D10DevSet.d3d10.DriverType == D3D10_DRIVER_TYPE_REFERENCE)
    t += (float)D3DX_PI * 0.0125f;
  else
    t += fElapsedTime;

  // Заполняем матрицу вращения
  D3DXMatrixRotationY( &g_World, t );
}
```

Эта функция отличается от той, которую мы использовали для вычисления мировой матрицы в *гл. 6*, только вызовом функции каркаса для получения параметров устройства и тем, что мы используем параметр `fElapsedTime` для определения времени, прошедшего с момента вывода предыдущего кадра.

Все подготовительные операции мы выполнили, переходим к выводу модели на экран. Откроем функцию `OnD3D10FrameRender()`, где происходит рисование кадра. Вывод модели на экран реализуется очень быстро. Будем вписывать строки после очистки шаблонного буфера глубины. Сначала мы передадим в шейдер рассчитанную мировую матрицу:

```
g_pWorldVariable->SetMatrix( (float*)&g_World );
```

После этого повторно свяжем объект входных данных с графическим конвейером, это необходимо потому, что вспомогательный класс для вывода текста при своей работе эту связь, образно говоря, нарушает. Можете потом закомментировать эту строку и посмотреть, как модель выводится без нее:

```
pd3dDevice->IASetInputLayout( g_pVertexLayout );
```

Теперь вызываем метод `CDXUTSDKMesh::Render`, именно он выведет нашу модель на экран, используя значения по умолчанию. Вызов метода выглядит так:

```
g_Mesh.Render(pd3dDevice, g_pTechnique, g_pTexResource);
```

При его вызове необходимо указать такие параметры:

- `pd3dDevice` — указатель на интерфейс устройства Direct3D 10;
- `g_pTechnique` — указатель на технику отображения, с помощью которой будет нарисована модель;
- `g_pTexResource` — указатель на переменную ресурса шейдера, через который с шейдером связывается накладываемая текстура.

Это все, что мы должны были добавить для появления модели на экране. Давайте посмотрим, как выглядит вся обновленная функция целиком:

```
void CALLBACK OnD3D10FrameRender( ID3D10Device* pd3dDevice,
                  double fTime, float fElapsedTime, void* pUserContext )
{
  // Очистка буфера визуализации и шаблонного буфера глубины
  float ClearColor[4] = { 0.176f, 0.196f, 0.667f, 0.0f };
  pd3dDevice->
     ClearRenderTargetView( DXUTGetD3D10RenderTargetView(), ClearColor );
  pd3dDevice->ClearDepthStencilView( DXUTGetD3D10DepthStencilView(),
                                     D3D10_CLEAR_DEPTH, 1.0, 0 );

  g_pWorldVariable->SetMatrix( (float*)&g_World );

  pd3dDevice->IASetInputLayout( g_pVertexLayout );

  g_Mesh.Render(pd3dDevice, g_pTechnique, g_pTexResource);

  g_pTxtHelper->Begin();
  g_pTxtHelper->SetInsertionPos( 5, 5 );
  g_pTxtHelper->SetForegroundColor( D3DXCOLOR( 1.0f, 1.0f, 1.0f, 1.0f ));
  g_pTxtHelper->DrawFormattedTextLine( L"Кадров в секунду: %.2f",
                                       DXUTGetFPS() );
  g_pTxtHelper->SetForegroundColor( D3DXCOLOR( 0.0f, 1.0f, 0.0f, 1.0f ));
  g_pTxtHelper->DrawTextLine( L"Нажмите <ESC> для выхода" );
  g_pTxtHelper->End();
}
```

Осталось обеспечить корректное освобождение памяти. За это, как мы помним, у нас отвечает функция `OnD3D10DestroyDevice()`. Добавим в нее строки для освобождения новых объектов:

```
void CALLBACK OnD3D10DestroyDevice( void* pUserContext )
{
```

```
  SAFE_RELEASE( g_pFont );
  SAFE_RELEASE( g_pSprite );
  SAFE_DELETE( g_pTxtHelper );
  SAFE_RELEASE( g_pVertexLayout );
  SAFE_RELEASE( g_pEffect );
  g_Mesh.Destroy();
}
```

Обратите внимание, что для удаления объекта, связанного с трехмерной моделью, нужно вызвать специальный метод: `CDXUTSDKMesh::Destroy`. Он автоматически освободит все ресурсы памяти, выделенные под трехмерную модель и ее текстуру.

Приложение для загрузки и вывода модели мы подготовили. Самое время заняться файлом эффектов и шейдерами. Давайте четко определим, что мы хотим получить с помощью шейдеров. Конечно, в первую очередь, это сама модель с наложенной текстурой. Чтобы как-то использовать координаты нормалей вершин, добавим к этому еще расчет освещения от направленного источника света. За основу можно взять файл эффектов, который мы использовали при выводе на экран куба с наложенной текстурой в *гл. 7.*

Создадим файл mesh.fx и начнем записывать в него исходный текст шейдеров. Сначала, как мы обычно делаем, определим структуры входных данных для вершинного и пиксельного шейдеров:

```
struct VS_INPUT
{
    float4 Pos  : POSITION;
    float3 Norm : NORMAL;
    float2 Tex  : TEXCOORD0;
};

struct PS_INPUT
{
    float4 Pos  : SV_POSITION;
    float3 Norm : TEXCOORD0;
    float2 Tex  : TEXCOORD1;
};
```

Структуры отличаются от ранее использованных только тем, что теперь в каждой из них мы применяем три вида координат.

Далее нам нужно определить режим работы сэмплера, который будет использован для наложения текстуры. Оставим его тем же, что мы уже исполь-

зовали, то есть линейное сэмплирование и режим адресации текстуры "обертывание":

```
SamplerState samLinear
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = Wrap;
    AddressV = Wrap;
};
```

При работе с DXUT придется дополнительно установить режимы работы с шаблонным буфером глубины и смешивания текстур. Это также вызвано тем, что вспомогательный класс для вывода текста отключает использование буфера глубины и устанавливает режим смешивания текстур с использованием прозрачности, а после завершения своей работы исходные значения не восстанавливает. Режимы, которые мы должны установить, устанавливаются таким же образом, как и режим работы сэмплера:

```
DepthStencilState EnableDepth
{
    DepthEnable = TRUE;
    DepthWriteMask = ALL;
    DepthFunc = LESS_EQUAL;
};

BlendState NoBlending
{
    AlphaToCoverageEnable = FALSE;
    BlendEnable[0] = FALSE;
};
```

Все описанные режимы мы применим позже, при описании техники отображения, а сейчас пока перейдем к функциям шейдеров. Для них нам потребуются глобальные переменные:

```
matrix World;
matrix View;
matrix Projection;

float3 vLightDir = float3( -0.577, 0.577, -0.577 );

Texture2D g_txDiffuse;
```

Новой здесь является переменная `vLightDir`, которая представляет собой глобальную константу, характеризующую вектор направления, в котором

распространяется свет от направленного источника (на самом деле, конечно, там содержится вектор противоположного направления, ведь при вычислении скалярного произведения нам нужно направление на источник света). На первый взгляд странные значения координат вектора объясняются тем, что для упрощения расчета освещенности здесь задается единичный вектор. Такой подход дает нам возможность не использовать при расчете освещенности функцию `normalize()` для этого вектора.

Как должна выглядеть функция вершинного шейдера? Она должна выполнять умножение координат вершины на все матрицы преобразований, умножение координат нормалей только на мировую матрицу, а текстурные координаты передавать без изменений. Вот текст функции, составленной по этим требованиям:

```
PS_INPUT VS( VS_INPUT Data )
{
    PS_INPUT Out;
    Out = (PS_INPUT)0;

    // Умножаем координаты вершины
    // на матрицы преобразований
    Out.Pos = mul (Data.Pos, World );
    Out.Pos = mul (Out.Pos, View);
    Out.Pos = mul (Out.Pos, Projection);

    // Умножаем нормали на мировую матрицу
    Out.Norm = mul (Data.Norm, World);

    // Текстурные координаты
    // передаем без изменений
    Out.Tex = Data.Tex;

    return Out;
}
```

В функции пиксельного шейдера для получения итогового цвета пиксела мы сначала вычисляем освещенность с помощью скалярного произведения вектора нормали на вектор направления на источник света. Затем мы определяем цвет пиксела, вызывая метод `Sample` текстурного объекта. Окончательное значение цвета пиксела определяется как результат умножения этих величин:

```
float4 PS_Mesh( PS_INPUT input ) : SV_Target

{
    float fLighting = saturate( dot( input.Norm, vLightDir ) );
```

```
    float4 finalColor = g_txDiffuse.Sample( samLinear, input.Tex ) *
                                        fLighting;
    finalColor.a = 1;
    return finalColor;
}
```

Остается только описать технику отображения, в ней мы должны установить все необходимые режимы:

```
technique10 RenderMesh
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, VS() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, PS_Mesh() ) );
        SetDepthStencilState( EnableDepth, 0 );
        SetBlendState( NoBlending, float4( 0.0f, 0.0f, 0.0f, 0.0f ),
                       0xFFFFFFFF );
    }
}
```

Полностью исходный текст файла mesh.fx представлен в листинге 10.1, просмотрим его еще раз.

Листинг 10.1

```
// Вывод на экран трехмерной модели
// Файл mesh.fx
//--------------------------------------------------------------------------
// Структуры данных
//--------------------------------------------------------------------------
struct VS_INPUT
{
    float4 Pos  : POSITION;
    float3 Norm : NORMAL;
    float2 Tex  : TEXCOORD0;
};

struct PS_INPUT
{
    float4 Pos  : SV_POSITION;
```

```
    float3 Norm : TEXCOORD0;
    float2 Tex  : TEXCOORD1;
};

SamplerState samLinear
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = Wrap;
    AddressV = Wrap;
};

DepthStencilState EnableDepth
{
    DepthEnable = TRUE;
    DepthWriteMask = ALL;
    DepthFunc = LESS_EQUAL;
};

BlendState NoBlending
{
    AlphaToCoverageEnable = FALSE;
    BlendEnable[0] = FALSE;
};
//--------------------------------------------------------------------------
// Глобальные переменные
//--------------------------------------------------------------------------
matrix World;
matrix View;
matrix Projection;

float3 vLightDir = float3(-0.577,0.577,-0.577);

Texture2D g_txDiffuse;

//--------------------------------------------------------------------------
// Функция вершинного шейдера
//--------------------------------------------------------------------------
PS_INPUT VS( VS_INPUT Data )
{
```

```
    PS_INPUT Out;
    Out = (PS_INPUT)0;

    // Умножаем координаты вершины
    // на матрицы преобразований
    Out.Pos = mul (Data.Pos, World );
    Out.Pos = mul (Out.Pos, View);
    Out.Pos = mul (Out.Pos, Projection);

    // Умножаем нормали на мировую матрицу
    Out.Norm = mul (Data.Norm, World);

    // Текстурные координаты
    // передаём без изменений
    Out.Tex = Data.Tex;

    return Out;
}

//--------------------------------------------------------------------------
// Функция пиксельного шейдера (рисует текстуру с учетом освещения)
//--------------------------------------------------------------------------
float4 PS_Mesh( PS_INPUT input ) : SV_Target

{
    float fLighting = saturate( dot( input.Norm, vLightDir ) );
    float4 finalColor = g_txDiffuse.Sample( samLinear, input.Tex ) *
                                    fLighting;
    finalColor.a = 1;
    return finalColor;
}

//--------------------------------------------------------------------------
// Technique
//--------------------------------------------------------------------------
technique10 RenderMesh
{
    pass P0
    {
```

```
        SetVertexShader( CompileShader( vs_4_0, VS() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, PS_Mesh() ) );
        SetDepthStencilState( EnableDepth, 0 );
        SetBlendState( NoBlending, float4( 0.0f, 0.0f, 0.0f, 0.0f ),
                       0xFFFFFFFF );
    }
}
```

Теперь у нас все готово. Пробуем скомпилировать и запустить проект. Мы должны увидеть на экране вращающийся куб, который мы сделали в 3DS Max (рис. 10.4).

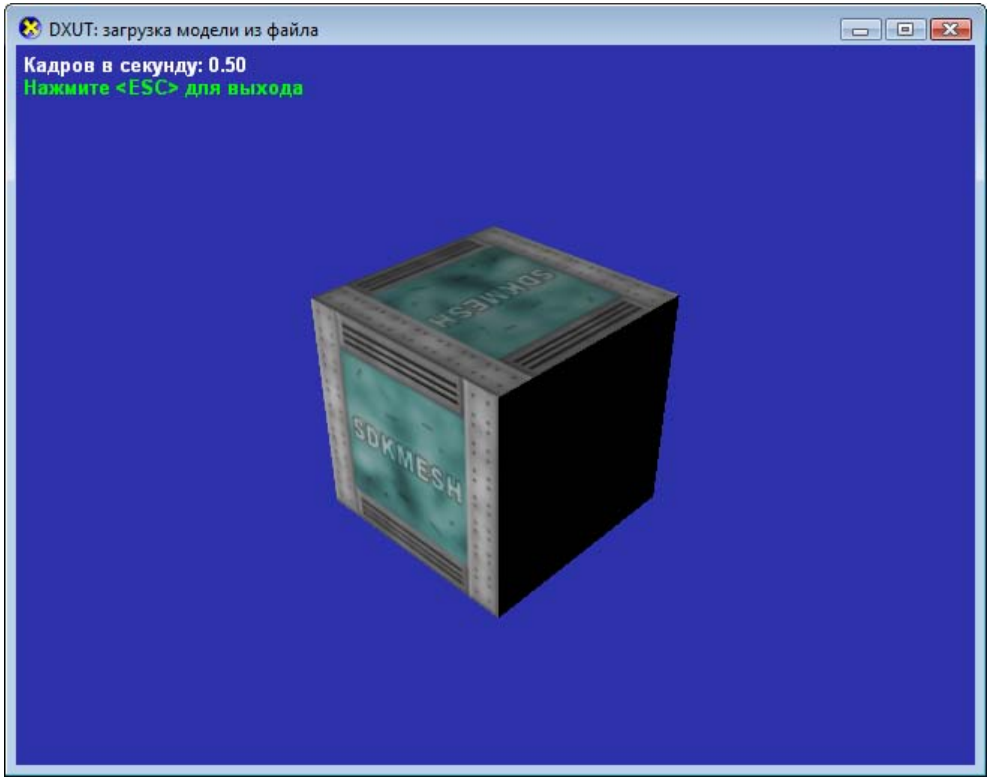

Рис. 10.4. Вывод трехмерной модели на экран

Вас, наверное, мучает вопрос: что будет, если не включить работу с шаблонным буфером глубины? Для вывода единственной модели в отдельности разницы мы не заметим. Но если добавить еще один объект, будет очевидно, что

сцена отображается неправильно. Чтобы убедиться в этом, достаточно вывести загруженную модель еще один раз, расположив ее подальше от камеры, за первым экземпляром модели. Сделать это очень просто, достаточно в функции `OnD3D10FrameRender()` добавить после вызова метода `CDXUTSDKMesh::Render` такие строки:

```
D3DXMatrixTranslation( &g_World, 0.0f, 0.05f, 2.0f );
g_pWorldVariable->SetMatrix( (float*)&g_World );
g_Mesh.Render( pd3dDevice, g_pTechniqueLt, g_pTexResource );
```

Если оставить режимы конвейера включенными, сцена будет отображаться корректно: второй объект будет частично закрываться первым (рис. 10.5).

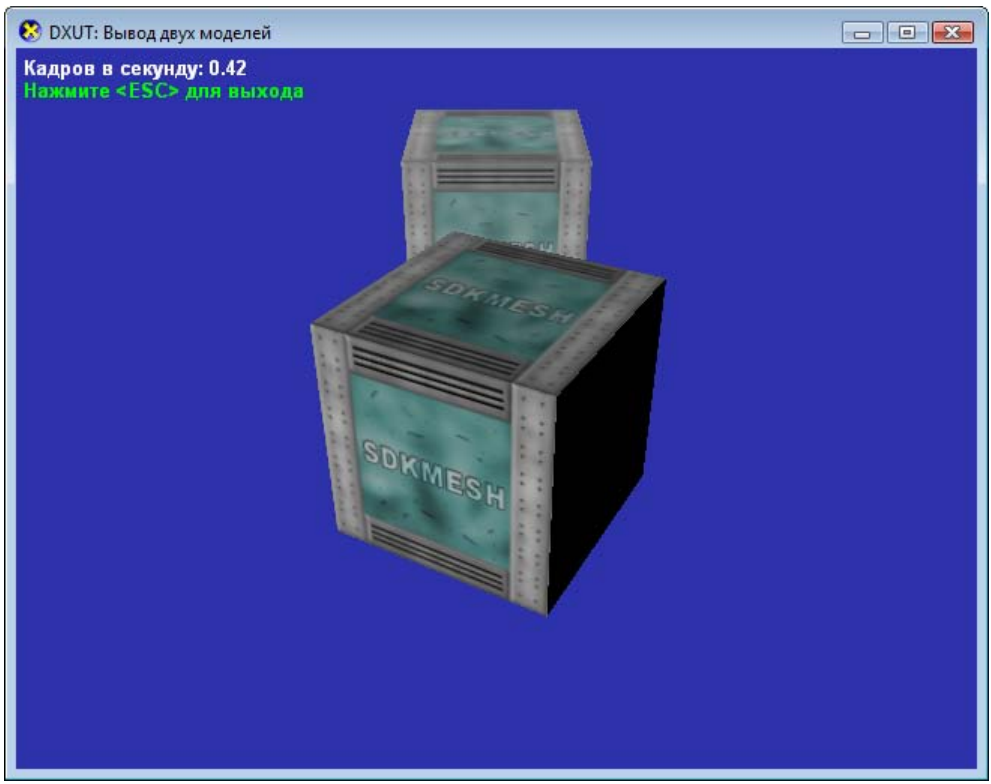

Рис. 10.5. Правильное отображение трехмерной сцены

Попробуйте закомментировать строки в описании техники отображения, где применяются режимы графического конвейера, и запустить программу с таким файлом эффектов. В результате мы получим уже совсем другое изображение (рис. 10.6).

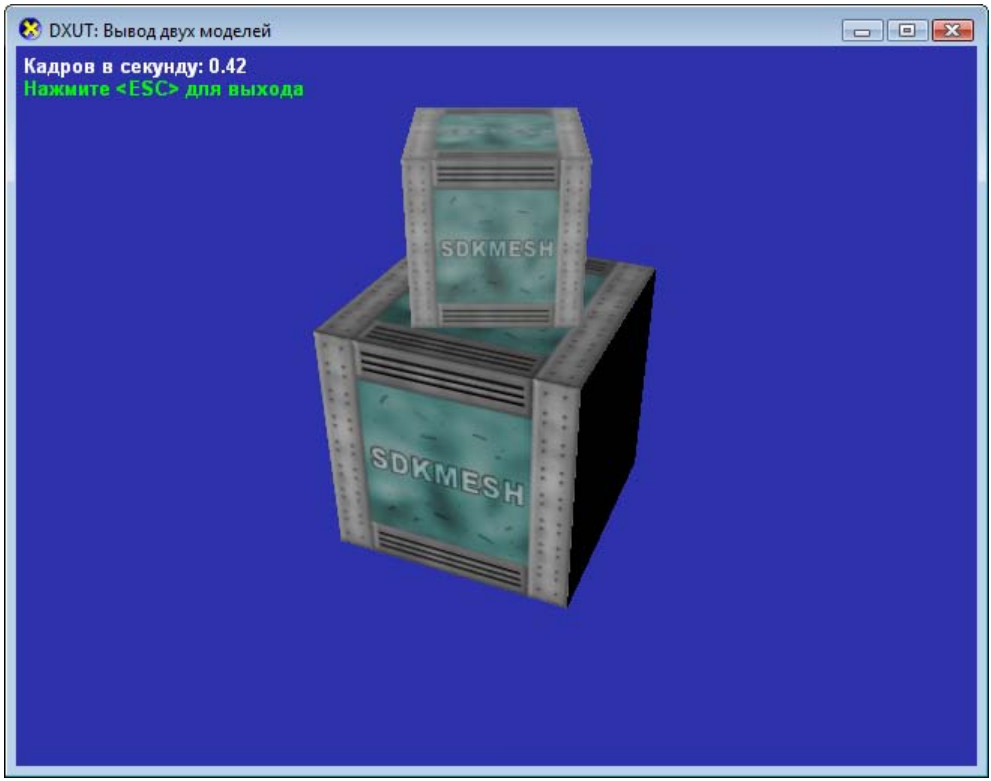

Рис. 10.6. Буфер глубины не используется: искаженное отображение трехмерной сцены

Может показаться, что второй объект меньше по размерам и что он стоит на первом, а это, как мы знаем, действительности не соответствует. Так получилось потому, что пикселы второго объекта в этом случае выводятся без учета расстояния от объекта до камеры и просто перекрывают уже имеющиеся в буфере пикселы первого объекта.

Файл box.sdkmesh, проект для загрузки из него модели, а также файл эффектов mesh.fx можно найти на компакт-диске в директории Glava10\DXUT_Mesh.

Камера для просмотра модели

Модели на экран мы теперь выводить умеем. Хотелось бы еще, чтобы можно было как-то управлять их поведением. Какое-нибудь сложное поведение с использованием искусственного интеллекта DXUT нам предоставить, конечно же, не сможет, но вращение модели с помощью мыши и изменение положения камеры мы можем реализовать без приложения титанических

усилий. Чтобы иметь такие возможности, нужно использовать класс `CModelViewerCamera`. Для его использования нужно подключить заголовочный файл DXUTcamera.h, а также добавить в проект файлы DXUTcamera.h и DXUTcamera.cpp.

Объявим экземпляр класса в глобальных переменных:

```
CModelViewerCamera          g_Camera;
```

Глобальные переменные для матриц вида и проекции (`g_View` и `g_Projection`) нам при работе с камерой не понадобятся, можно их удалить из программы. Теперь нам необходимо инициализировать камеру, то есть задать ее матрицу вида. Матрицу вида камера рассчитает сама, от нас требуется только передать ей координаты двух точек: точки, в которой располагается камера, и точки, в которую она направлена. Инициализация матрицы вида у нас производится в функции `OnD3D10CreateDevice()`, в ней нам нужно исправить лишь одну строчку в самом конце:

```
// Инициализируем камеру
D3DXVECTOR3 Eye( 0.0f, 2.0f, -3.0f );
D3DXVECTOR3 At( 0.0f, 0.0f, 0.0f );
g_Camera.SetViewParams( &Eye, &At );
```

Настроим реакцию камеры на изменение размеров окна. Открываем функцию `OnD3D10ResizedSwapChain()`, она будет теперь выглядеть следующим образом:

```
HRESULT CALLBACK OnD3D10ResizedSwapChain( ID3D10Device* pd3dDevice,
                        IDXGISwapChain *pSwapChain,
                        const DXGI_SURFACE_DESC* pBackBufferSurfaceDesc,
                        void* pUserContext )
{
// Устанавливаем параметры матрицы проекции
  float fAspect = (float)( pBackBufferSurfaceDesc->Width )/
                           (float)( pBackBufferSurfaceDesc->Height );
  g_Camera.SetProjParams( D3DX_PI * 0.25f, fAspect, 0.1f, 100.0f );
  g_Camera.SetWindow( pBackBufferSurfaceDesc->Width,
                      pBackBufferSurfaceDesc->Height );
  g_Camera.SetButtonMasks( MOUSE_LEFT_BUTTON, MOUSE_WHEEL,
                           MOUSE_RIGHT_BUTTON );

  return S_OK;
}
```

Сначала мы рассчитываем соотношение сторон окна, после этого вызываем метод `SetProjParams` класса камеры, который устанавливает матрицу проекции.

Его параметры идентичны параметрам функции `D3DXMatrixPerspectiveFovLH()`, (на самом деле, матрица проекции внутри этого метода заполняется именно с помощью этой функции).

После установки матрицы проекции мы вызываем метод `SetWindow`, он нужен для правильного отображения сцены вращающейся камерой. Ему мы передаем новые значения ширины и высоты окна.

Метод `SetButtonMasks` отвечает за управление моделью и камерой с помощью мыши, в качестве параметров указываются клавиши мыши, с которыми нужно связать вращение модели, приближение/удаление модели и вращение камеры соответственно. У нас установлено вращение модели перемещением мыши при нажатой левой кнопке, приближение/удаление осуществляется с помощью колесика мыши, а камера вращается мышью при нажатой правой клавише.

Расчет мировой матрицы при наличии камеры также будет проводиться несколько иначе. Поскольку мы можем менять положение модели в пространстве с помощью камеры, мы будем получать от нее мировую матрицу, которая отражает изменения, внесенные с помощью камеры, а уже потом "накладывать" на нее вращение вокруг оси Y. Работа с мировой матрицей попрежнему остается в функции `OnFrameMove()`, рассмотрим ее полностью:

```
void CALLBACK OnFrameMove( double fTime, float fElapsedTime,
                           void* pUserContext )
{
  // Обновляем положение камеры
  g_Camera.FrameMove(fElapsedTime);

  // Получаем параметры устройства Direct3D
  DXUTDeviceSettings D3D10DevSet;
  D3D10DevSet = DXUTGetDeviceSettings();
  static float t = 0.0f;

  // Устанавливаем угол поворота в зависимости от типа
  //драйвера устройства
  if (D3D10DevSet.d3d10.DriverType == D3D10_DRIVER_TYPE_REFERENCE)

    t += (float)D3DX_PI * 0.0125f;
  else
    t += fElapsedTime;

  // Заполняем матрицу вращения
```

```
  D3DXMATRIX mRotY;
  D3DXMatrixRotationY( &mRotY, t );

  // Устанавливаем мировую матрицу
  g_World = *g_Camera.GetWorldMatrix();
  g_World = g_World * mRotY;
}
```

В первой строке мы вызываем метод камеры `FrameMove`, который отвечает за изменение матрицы вида в зависимости от действий пользователя (вращение модели, изменение положения камеры и т. д.). Определение типа драйвера устройства и назначение угла вращения производятся так же, как раньше. Отличие состоит в том, что мы заполняем отдельную матрицу вращения `mRotY`. В матрицу `g_World` мы извлекаем мировую матрицу от камеры с помощью метода `GetWorldMatrix`, а потом объединяем ее с матрицей вращения, перемножив их между собой.

Так как все матрицы, кроме мировой, мы должны получить от экземпляра класса камеры, нам потребуется ввести в функцию `OnD3D10FrameRender()` строки, где происходит передача этих матриц соответствующим переменным шейдера. Выглядеть эта передача будет следующим образом:

```
g_pProjectionVariable->SetMatrix( (float*)g_Camera.GetProjMatrix() );
g_pViewVariable->SetMatrix( (float*)g_Camera.GetViewMatrix() );
g_pWorldVariable->SetMatrix( (float*)&g_World );
```

Новая версия функции `OnD3D10FrameRender()` должна выглядеть следующим образом (обратите внимание, что мировую матрицу мы по-прежнему передаем из глобальной переменной):

```
void CALLBACK OnD3D10FrameRender( ID3D10Device* pd3dDevice,
                                  double fTime, float fElapsedTime,
                                  void* pUserContext )
{
  // Очистка буфера визуализации и шаблонного буфера глубины
  float ClearColor[4] = { 0.176f, 0.196f, 0.667f, 0.0f };
  pd3dDevice->
     ClearRenderTargetView( DXUTGetD3D10RenderTargetView(), ClearColor );
  pd3dDevice->ClearDepthStencilView( DXUTGetD3D10DepthStencilView(),
                                     D3D10_CLEAR_DEPTH, 1.0, 0 );

  // Передаем в шейдеры значения переменных
  g_pProjectionVariable->SetMatrix( (float*)g_Camera.GetProjMatrix() );
```

```
  g_pViewVariable->SetMatrix( (float*)g_Camera.GetViewMatrix() );
  g_pWorldVariable->SetMatrix( (float*)&g_World );

  pd3dDevice->IASetInputLayout( g_pVertexLayout );

  g_Mesh.Render(pd3dDevice, g_pTechnique, g_pTexResource);

  g_pTxtHelper->Begin();
  g_pTxtHelper->SetInsertionPos( 5, 5 );
  g_pTxtHelper->SetForegroundColor( D3DXCOLOR( 1.0f, 1.0f, 1.0f, 1.0f ));
  g_pTxtHelper->
        DrawFormattedTextLine( L"Кадров в секунду: %.2f", DXUTGetFPS() );
  g_pTxtHelper->
        SetForegroundColor( D3DXCOLOR( 0.0f, 1.0f, 0.0f, 1.0f ) );
  g_pTxtHelper->DrawTextLine( L"Нажмите <ESC> для выхода" );
  g_pTxtHelper->End();
}
```

Все уже практически готово, осталось обеспечить передачу поступающих в очередь окна сообщений Windows камере, чтобы она могла реагировать на изменение положения курсора мыши и нажатие кнопок. Для этого нам нужно задействовать функцию `MsgProc()`, в нее мы поместим вызов метода `HandleMessages`, который отвечает за обработку сообщений. Методу мы передаем те же параметры, что и функции `WndProc()` из нашего первого приложения Windows, остальные параметры функции `MsgProc()` мы пока не используем:

```
LRESULT CALLBACK MsgProc( HWND hWnd, UINT uMsg, WPARAM wParam,
                          LPARAM lParam, bool* pbNoFurtherProcessing,
                          void* pUserContext )
{
  // Передаем поступившие сообщения на обработку камере
  g_Camera.HandleMessages( hWnd, uMsg, wParam, lParam );
  return 0;
}
```

"Внедрение" камеры в программу на этом закончено. Откомпилируем и запустим проект. Результат работы программы выглядит точно так же, как и у предыдущего проекта. Проверим работу камеры. Попробуйте повернуть модель, передвигая мышь, при этом удерживая нажатой левую кнопку: куб должен отреагировать и изменить свое положение (рис. 10.7).

Вращением колесика мыши можно придвинуть модель к экрану либо уменьшить ее практически до размеров пиксела экрана. Положение самой камеры, можно менять, перемещая мышь с нажатой правой кнопкой.

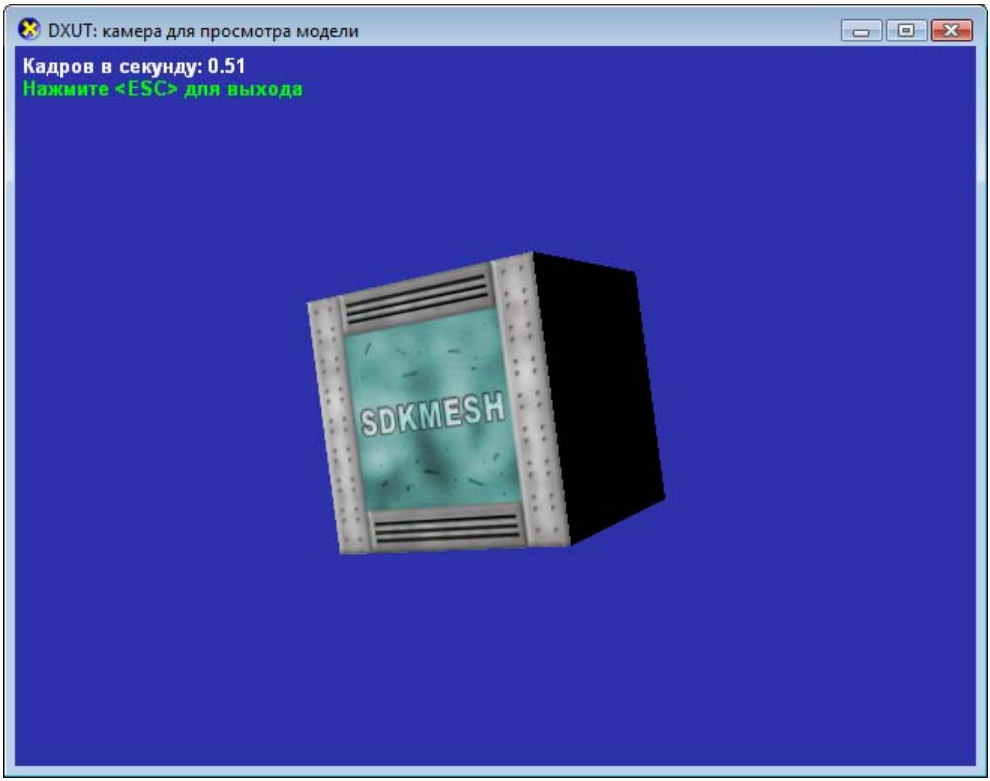

Рис. 10.7. Использование возможностей камеры для вращения модели

Замечание

Если у вас используется драйвер устройства с программной эмуляцией графического конвейера (Reference device), увидеть изменение положения модели будет довольно непросто из-за медленной работы. Немного улучшить положение можно, уменьшив размеры окна: при этом количество кадров в секунду возрастет, и движение модели станет более плавным.

Соответствующий проект, использующий класс камеры, находится на компакт-диске в директории Glava10\DXUT_Camera.

Глава 11

Пользовательский интерфейс

Как известно, при разработке программного обеспечения много времени уходит на создание интерфейса пользователя. Написать "с нуля" библиотеку, реализующую функционал элементов интерфейса Windows, является непростой задачей даже для опытного программиста. Но при использовании DXUT об интерфейсе можно больше не беспокоиться: он, что называется, "входит в комплект". Нам нужно только научиться применять то, что уже сделано профессиональными программистами.

Обзор элементов интерфейса

Давайте посмотрим, какие элементы DXUT может нам предоставить. Начнем с самого простого элемента, затем постепенно переходя к более сложным в программировании. В нашу программу мы будем их добавлять именно в такой последовательности.

Статичный текст (Static text)

Самым простым элементом интерфейса, наверное, является статичный текст. Он во многом похож на тот текст, который мы выводили с помощью класса `CDXUTTextHelper`. При помощи статичного текста можно выводить текстовые сообщения и значения переменных. Разница состоит в оформлении текста, здесь он выводится на экран "с тенью" (рис. 11.1).

Рис. 11.1. Статичный текст

Кнопка (Button)

Кнопка играет ту же самую роль, что и кнопка стандартного интерфейса Windows: реагирует на щелчок мыши, выполняя при этом какое-то действие. Можно, например, по нажатию кнопки переключить программу в полноэкранный режим, вызвать диалоговое окно с параметрами вывода графики или сделать другое полезное дело. Как кнопка выглядит на экране, можно увидеть на рис. 11.2.

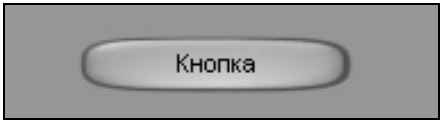

Рис. 11.2. Кнопка

Флажок (Checkbox)

Флажок позволяет включить или выключить выполнение какого-либо действия. Например, с помощью флажка можно указывать наличие или отсутствие в трехмерной сцене источника света, необходимость проработки отбрасываемых теней и т. д. На экране флажок выглядит так, как показано на рис. 11.3.

Рис. 11.3. Флажок

Поле ввода (Editbox)

Поле ввода дает возможность пользователю ввести однострочный текст. Его можно использовать, чтобы получить имя игрока для таблицы рекордов, для ввода пароля, пути к файлу на диске, адреса сервера в Интернете, а также для многого другого. Выглядит поле ввода практически так же, как его аналог из стандартного интерфейса Windows (рис. 11.4).

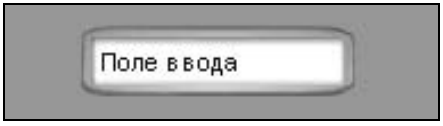

Рис. 11.4. Поле ввода

Имеется также версия поля ввода с поддержкой IME (Input Method Editor — редактор метода ввода). Данный элемент предоставляет возможность ввода символов, отсутствующих на клавиатуре. Чаще всего таким образом вводят текст, содержащий корейские, японские или китайские иероглифы. От уже рассмотренного поля ввода версия с поддержкой IME внешне отличается только наличием индикатора раскладки клавиатуры (рис. 11.5).

Рис. 11.5. Поле ввода с поддержкой IME

Выпадающий список (Combobox)

Этот элемент позволяет выбрать какой-либо пункт из выпадающего списка. При помощи выпадающего списка можно реализовать выбор языка, на котором будут выводиться сообщения программы, предоставить для выбора список режимов работы с различным разрешением и т. д. Выпадающий список, в котором содержится три элемента, выглядит, как показано на рис. 11.6.

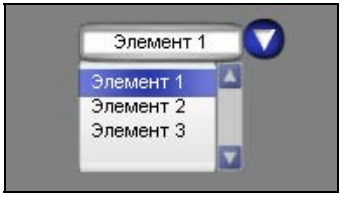

Рис. 11.6. Выпадающий список

Список (Listbox)

При помощи списка можно выбрать один или несколько из имеющихся элементов. Так можно организовать, например, выбор карты и сервера для сетевой игры. Как на экране выглядит список, содержащий три элемента, можно увидеть на рис. 11.7.

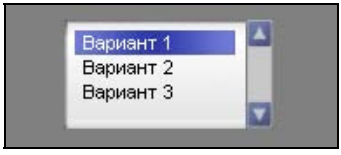

Рис. 11.7. Список

Ползунок (Slider)

При помощи ползунка можно установить значение какого-нибудь параметра, ориентируясь на его максимальное и минимальное значение. Таким образом, можно, например, задать скорость вращения модели, интенсивность света, уровень громкости звука… Численное отображение значения с помощью ползунка не предусмотрено, однако эту возможность очень легко реализовать, используя статичный текст. На рис. 11.8 представлен внешний вид ползунка.

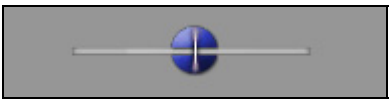

Рис. 11.8. Ползунок

Переключатели (Radiobutton)

Эти элементы используются в том случае, когда пользователь должен выбрать только один из предложенных вариантов (рис. 11.9), например, использование программой либо полноэкранного, либо оконного режима.

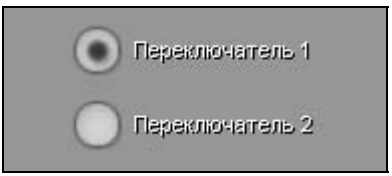

Рис. 11.9. Переключатель

Добавляем интерфейс в программу

Теперь мы знаем "в лицо" все элементы интерфейса. Нам осталось научиться добавлять каждый из них в свою программу и обращаться с ними. Рассмотрим общий подход к работе с элементами пользовательского интерфейса DXUT.

Общий подход

Для использования интерфейса пользователя DXUT нам потребуется ввести в программу как минимум два новых объекта: это диалоговое окно и менеджер ресурсов диалоговых окон:

- Диалоговому окну принадлежат все элементы интерфейса. Оно отвечает за обработку данных от пользователя и за отображение элементов на экране.

- Менеджер ресурсов диалоговых окон отвечает за управление общими ресурсами, которые используются диалоговыми окнами (шрифты, текстуры).

Пусть вся эта терминология вас не пугает, нам не придется вникать, как происходит управление ресурсами, это все скрыто от нас внутри каркаса DXUT. Перейдем к практической стороне вопроса и познакомимся с тем, какие действия необходимо выполнить, чтобы добавить интерфейс в свою программу.

В первую очередь нужно подключить заголовочный файл DXUTgui.h и, соответственно, добавить в проект файлы DXUTgui.h и DXUTgui.cpp.

Добавить глобальные переменные, представляющие собой диалоговое окно (`CDXUTDialog`) и менеджер ресурсов (`CDXUTDialogResourceManager`). Определить константы, содержащие идентификаторы элементов интерфейса, которые мы собираемся использовать.

Инициализировать элементы интерфейса. На этом этапе определяется, какие элементы используются и где они расположены. Поскольку эту инициализацию необходимо провести до создания устройства Direct3D, для нее создается отдельная функция, которую мы вызываем из функции `wWinMain()`, ее обычно размещают перед вызовом функции `DXUTInit()`.

Инициализировать менеджер ресурсов, для этого у него есть специальный метод: `CDXUTDialogResourceManager::OnD3D10CreateDevice`. Делается это уже после создания устройства Direct3D, поэтому этот метод мы вызываем из функции `OnD3D10CreateDevice()`.

Обеспечить прорисовку элементов интерфейса, для этого в функцию `OnD3D10FrameRender()` добавляем вызов соответствующего метода диалогового окна: `CDXUTDialog::OnRender`.

Для обеспечения визуальной ответной реакции на действия пользователя добавить в функцию `MsgProc()` вызов методов для обработки сообщений менеджера ресурсов и диалогового окна: методы `CDXUTDialogResourceManager::MsgProc` и `CDXUTDialog::MsgProc`. Чтобы обеспечить функциональную реакцию (например, действия по нажатию кнопки), нужно ввести специальную функцию, будем называть ее `OnGUIEvent()`.

Для правильного удаления объектов менеджера ресурсов при изменении размеров окна добавить в функцию `OnD3D10ReleasingSwapChain()` вызов специального метода менеджера ресурсов:
`CDXUTDialogResourceManager::OnD3D10ReleasingSwapChain`.

Для восстановления освобожденных объектов и передачи информации о новых размерах окна в менеджер ресурсов использовать предназначенный именно для этого метод `CDXUTDialogResourceManager::OnD3D10ResizedSwapChain`. При необходимости также можно скорректировать положение элементов интер-

фейса на экране после изменения размеров окна, что также нужно сделать в функции `OnD3D10ResizedSwapChain()`.

Для корректного освобождения памяти при выходе из программы в функцию `OnD3D10DestroyDevice()` добавить вызов метода, освобождающего память, которая выделена для элементов интерфейса пользователя:
`CDXUTDialogResourceManager::OnD3D10DestroyDevice`.

После прочтения этого списка у вас, наверное, возникло ощущение, что лучше отказаться от интерфейса вообще, чем во всем этом разбираться. Спокойствие, только спокойствие: все только выглядит сложным, настроив работу с интерфейсом один раз, для первого элемента, добавлять последующие элементы будет уже значительно проще. Написав первую программу с использованием интерфейса, вы сами в этом убедитесь. Поэтому давайте сразу перейдем к практике.

Добавляем статичный текст

Сформулируем задание: составить программу, использующую статичный текст. Текст разместить в нижней части окна посередине, обеспечить корректировку положения текста при изменении размеров окна и переключении в полноэкранный режим.

Приступим к выполнению. Сделаем копию нашего последнего проекта, выводящего модель на экран, где мы использовали камеру. Добавим к проекту файлы DXUTgui.h и DXUTgui.cpp, подключим заголовочный файл, чтобы иметь возможность использовать интерфейс:

```
#include "DXUTgui.h"
```

Объявим глобальные переменные для менеджера ресурсов и диалогового окна:

```
CDXUTDialogResourceManager  g_DialogResourceManager;
CDXUTDialog                 g_UserInterface;
```

Определим константу, представляющую собой идентификатор элемента интерфейса, в нашем случае — статичного текста:

```
#define IDC_STATICTEXT          1
```

Теперь необходимо выполнить инициализацию менеджера ресурсов и ввод элементов интерфейса. Добавим предварительное объявление функции инициализации:

```
void InitApp();
```

> ***Замечание***
>
> Вообще говоря, здесь можно было обойтись без предварительного объявления, оно необходимо только в том случае, если вы добавите текст функции `InitApp()` после функции `wWinMain()`. То есть опишете функцию после ее вызова.

После этого напишем текст функции:

```
void InitApp()
{
  // Инициализируем интерфейс
  g_UserInterface.Init( &g_DialogResourceManager );

  g_UserInterface.AddStatic( IDC_STATICTEXT, L"Статичный текст",
                             200, 450, 90, 24 );
}
```

Что происходит в этой функции? В первой строке мы вызываем метод `Init` диалогового окна, передавая в качестве параметра указатель на объект менеджера ресурсов. Этот метод регистрирует диалоговое окно в менеджере ресурсов и выполняет инициализацию элементов интерфейса по умолчанию (задает их внешний вид).

Во второй строке мы вызываем метод `CDXUTDialog::AddStatic`, чтобы добавить статичный текст в диалоговое окно. Рассмотрим прототип этого метода:

```
HRESULT AddStatic(
 int ID,
 LPCWSTR strText,
 int x,
 int y,
 int width,
 int height,
 bool bIsDefault,
 CDXUTStatic** ppCreated
 )
```

Чтобы создать статичный текст, мы должны передать следующие параметры:

- `ID` — идентификатор элемента интерфейса, представляющий собой своего рода "инвентарный номер", используя который можно быстро получить доступ к свойствам и методам элемента;
- `strText` — текст, который должен содержать элемент;
- `x` и `y` —положение верхнего левого угла элемента относительно осей X и Y, соответственно;
- `width` и `height` — ширина и высота элемента в пикселях.
- `bIsDefault` — флаг, определяющий, является ли элемент элементом по умолчанию (установлен ли сразу на него фокус ввода), значение по умолчанию — `FALSE`;

- `ppCreated` — адрес указателя, который может быть установлен на создаваемый элемент, по умолчанию установлено значение `NULL`, и указатель не устанавливается.

Этот набор параметров является базовым для всех элементов интерфейса, как мы увидим впоследствии. Несколько слов нужно сказать об используемой системе координат. Вообще, при установке положения элементов интерфейса нужно использовать систему координат, связанную с диалоговым окном. Но по умолчанию, если размеры окна не указаны, размеры диалогового окна принимаются равными соответствующим размерам вторичного буфера. Поэтому оконная система координат приложения и система координат диалогового окна в данном случае также совпадают. Еще нужно отметить, что по умолчанию само диалоговое окно полностью прозрачно и ничего, кроме элементов интерфейса, отображать не будет.

> ***Замечание***
>
> Самое интересное, что координаты положения элемента, заданные при инициализации, могут не играть никакой роли. Это имеет место в том случае, если происходит корректировка положения элементов интерфейса в функции `OnD3D10ResizedSwapChain()`, которая, как вы помните, вызывается сразу за функцией `OnD3D10CreateDevice()`. Во время корректировки положение элемента будет пересчитано, и на том месте, которое мы указали при инициализации, его может не оказаться.

Не забываем добавить вызов функции `InitApp()` в функцию `wWinMain()` перед началом инициализации DXUT (перед функцией `DXUTInit()`).

Следующим шагом мы должны инициализировать менеджер ресурсов. Открываем функцию `OnD3D10CreateDevice()` и после объявления переменной `hr` записываем следующую строку:

```
V_RETURN(g_DialogResourceManager.OnD3D10CreateDevice( pd3dDevice ));
```

Методу передается указатель на интерфейс созданного устройства Direct3D 10. Внутри метода производится создание эффекта для прорисовки интерфейса, создание шрифтов, текстур и другая подготовка данных.

Переходим к функции `OnD3D10FrameRender()`, в которой нам нужно обеспечить вывод элементов на экран. После прорисовки трехмерной модели вводим вот такой текст:

```
g_UserInterface.OnRender(fElapsedTime);
```

Методу передается время, прошедшее с прорисовки последнего кадра, оно необходимо для анимации элементов интерфейса, например, при наведении курсора мыши. У нас пока имеется один статичный текст, и никакой анимации для него не предусмотрено. Чтобы элемент "знал", что на него навели курсор или щелкнули по нему мышью, диалоговое окно должно получить

соответствующее сообщение от системы. Открываем функцию `MsgProc()` и записываем вызовы методов для обработки сообщений. Вот как выглядит эта функция целиком:

```
LRESULT CALLBACK MsgProc( HWND hWnd, UINT uMsg, WPARAM wParam,
                          LPARAM lParam, bool* pbNoFurtherProcessing,
                          void* pUserContext )
{
  *pbNoFurtherProcessing =
          g_DialogResourceManager.MsgProc( hWnd, uMsg, wParam, lParam );
  if( *pbNoFurtherProcessing )
    return 0;

  // Обработка сообщений элементами интерфейса
  *pbNoFurtherProcessing =
                  g_UserInterface.MsgProc( hWnd, uMsg, wParam, lParam );
  if( *pbNoFurtherProcessing )
    return 0;

  // Передаем поступившие сообщения на обработку камере
  g_Camera.HandleMessages( hWnd, uMsg, wParam, lParam );
  return 0;
}
```

Параметр `pbNoFurtherProcessing` указывает, нужна ли дальнейшая обработка поступившего сообщения. Если очередной метод вернул значение `FALSE`, значит, сообщение обработано, дальнейшая обработка не требуется, будет произведен выход из функции `MsgProc()`. Вызов метода для обработки сообщений менеджера ресурсов связан только с элементами, поддерживающими IME. За обработку сообщений для всех элементов интерфейса целиком отвечает метод диалогового окна.

После того как мы реализовали обработку сообщений, определим действия, которые происходят при изменении размеров окна. Сначала рассмотрим, что происходит при освобождении цепочки переключений, здесь все просто, мы вызываем соответствующий метод менеджера ресурсов:

```
void CALLBACK OnD3D10ReleasingSwapChain( void* pUserContext )
{
  g_DialogResourceManager.OnD3D10ReleasingSwapChain();
}
```

После того как размеры окна изменились и каркас установил новые размеры вторичного буфера, вызывается функция `OnD3D10ResizedSwapChain()`, в ко-

торой мы должны поместить статичный текст по центру нового окна. Приведем функцию полностью:

```
HRESULT CALLBACK OnD3D10ResizedSwapChain( ID3D10Device* pd3dDevice,
                          IDXGISwapChain *pSwapChain,
                          const DXGI_SURFACE_DESC* pBackBufferSurfaceDesc,
                          void* pUserContext )
{
  HRESULT hr;

  V_RETURN( g_DialogResourceManager.OnD3D10ResizedSwapChain( pd3dDevice,
                                                  pBackBufferSurfaceDesc ) );

  // Устанавливаем параметры матрицы проекции
  float fAspect = (float)( pBackBufferSurfaceDesc->Width )/
                               (float)( pBackBufferSurfaceDesc->Height );
  g_Camera.SetProjParams( D3DX_PI * 0.25f, fAspect, 0.1f, 100.0f );
  g_Camera.SetWindow( pBackBufferSurfaceDesc->Width,
                      pBackBufferSurfaceDesc->Height );
  g_Camera.SetButtonMasks( MOUSE_LEFT_BUTTON, MOUSE_WHEEL,
                           MOUSE_RIGHT_BUTTON );

  // Передвигаем элементы интерфейса, подстраиваемся под новые
  // размеры окна
  // Размещаем элемент посередине окна:
  // берем ширину нашего элемента
  int StaticWidth =
        g_UserInterface.GetControl( IDC_STATICTEXT )->m_width;
  int WindowWidth = pBackBufferSurfaceDesc->Width;
  // Зная ширину элемента и ширину окна
  // рассчитываем координату X : (WindowWidth-StaticWidth)/2
  g_UserInterface.GetControl( IDC_STATICTEXT )->
                    SetLocation( (WindowWidth-StaticWidth)/2,
                                 pBackBufferSurfaceDesc->Height-30 );

  return S_OK;
}
```

В самом начале функции мы вызываем метод менеджера ресурсов `OnD3D10ResizedSwapChain`, ему передается указатель на устройство Direct3D 10 и структура с описанием вторичного буфера. После этого идут уже знакомые

нам операции с матрицей проекции и камерой. Вслед за этим мы устанавливаем новое положение элемента с помощью его метода `SetLocation`. Координату X рассчитываем как половину разницы между новой шириной окна и шириной элемента интерфейса. Координату Y элемента определяем как отстоящую на 30 пикселов вверх от нижней границы окна. Чтобы получить доступ к методам элемента, нужно вызвать метод `GetControl` диалогового окна, указав в качестве параметра идентификатор нужного элемента интерфейса.

Нам осталось выполнить последний пункт — обеспечить корректное освобождение памяти при выходе из программы. Идем в функцию `OnD3D10DestroyDevice()` и приводим ее к следующему виду:

```
void CALLBACK OnD3D10DestroyDevice( void* pUserContext )
{
  g_DialogResourceManager.OnD3D10DestroyDevice();
  DXUTGetGlobalResourceCache().OnDestroyDevice();
  SAFE_RELEASE( g_pFont );
  SAFE_RELEASE( g_pSprite );
  SAFE_DELETE( g_pTxtHelper );
  SAFE_RELEASE( g_pVertexLayout );
  SAFE_RELEASE( g_pEffect );
  g_Mesh.Destroy();
}
```

С вызовом метода менеджера ресурсов `OnD3D10DestroyDevice`, наверное, все ясно, более или менее. А вот что это такое в строке сразу после него? Все дело в том, что метод освобождает ресурсы, которые создал менеджер ресурсов, но он не очищает кэш ресурсов DXUT. В непонятной строке как раз и происходит очистка кэша.

На этом мы закончили доработку программы для использования интерфейса. Откомпилируем проект, запустим его и посмотрим на результат своего труда. Как видно, в нижней части окна (30 пикселов от нижней границы, помните?) появилась надпись "Статичный текст" (рис. 11.10). Попробуем изменить размеры окна, — надпись сохранит свое положение по центру. Готовый проект с использованием статичного текста можно найти на компакт-диске в директории Glava11\DXUT_Static.

При доработке программы мы выполнили все пункты плана. Не затронули мы пока только функцию `OnGUIEvent()`, отвечающую за привязку действий к элементам интерфейса. Но здесь она нам и не была нужна: статичный текст используется только для отображения информации и не принимает данных

от пользователя. Этот пробел мы восполним, когда добавим в программу кнопку для переключения в полноэкранный режим.

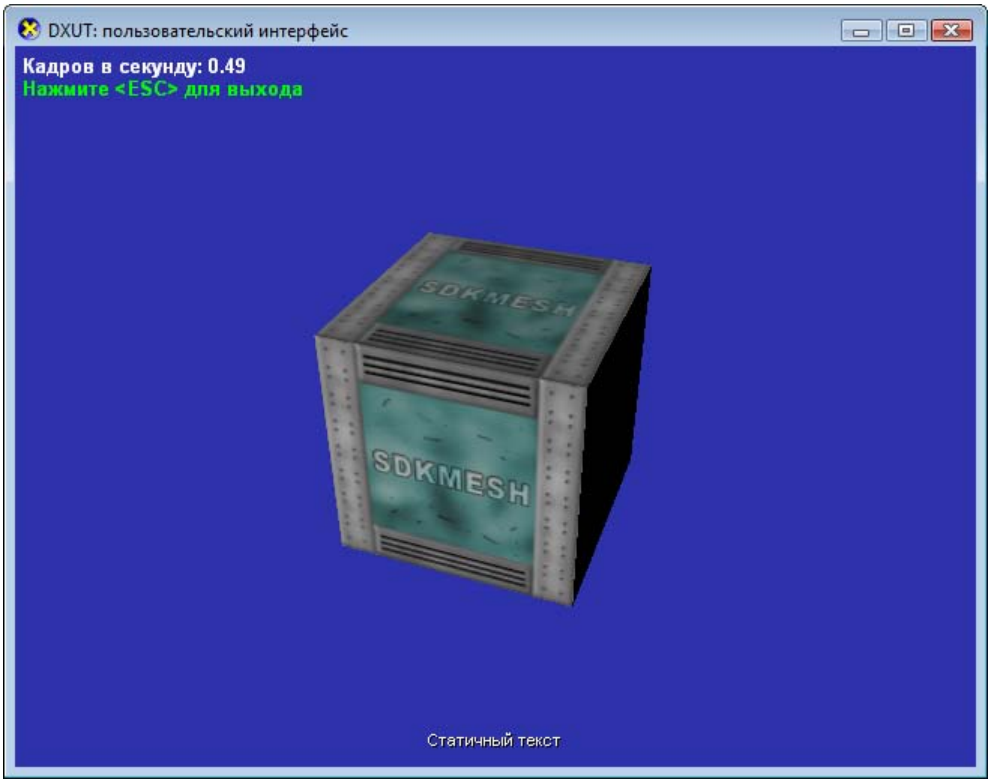

Рис. 11.10. Интерфейс пользователя: статичный текст

Добавляем кнопку

Как говорилось ранее, добавлять последующие элементы будет проще, поскольку главную работу по подготовке работы с интерфейсом мы уже сделали. Поставим себе следующую задачу: добавить в программу кнопку, переключающую программу в полноэкранный режим и обратно, при этом обеспечить расположение кнопки в правом верхнем углу окна и изменение текста на кнопке в зависимости от режима, который устанавливается при нажатии на нее.

Приступим к реализации задачи. В первую очередь определим идентификатор для нового элемента интерфейса:

```
#define IDC_TOGGLEFULLSCREEN    2
```

После этого переходим в функцию инициализации интерфейса, где добавляем кнопку и устанавливаем функцию для обработки событий интерфейса. Функция инициализации после этого примет следующий вид:

```
void InitApp()
{
  // Инициализируем интерфейс
  g_UserInterface.Init( &g_DialogResourceManager );

  // Установим функцию для обработки событий
  g_UserInterface.SetCallback( OnGUIEvent );

  g_UserInterface.AddStatic( IDC_STATICTEXT, L"Статичный текст",
                             200, 450, 90, 24 );
  g_UserInterface.AddButton( IDC_TOGGLEFULLSCREEN,
                             L"Полноэкранный режим", 480, 25, 140, 25 );
}
```

После инициализации диалогового окна мы связываем с ним функцию `OnGUIEvent()`, в которой будем обрабатывать нажатие на кнопку. Поскольку эту функцию мы еще не описали, а уже использовали в программе, нужно добавить ее предварительное объявление. Для этого впечатаем ниже предварительного объявления функции `InitApp()` такой текст:

```
void    CALLBACK OnGUIEvent( UINT nEvent, int nControlID,
                             CDXUTControl* pControl, void* pUserContext );
```

Саму кнопку мы добавляем при помощи метода `AddButton`. Он во многом похож на метод `AddStatic`, здесь даже не видно разницы в параметрах. Однако некоторые различия присутствуют, чтобы их выявить, рассмотрим прототип метода `AddButton` диалогового окна:

```
HRESULT AddButton(
  int ID,
  LPCWSTR strText,
  int x,
  int y,
  int width,
  int height,
  UINT nHotkey,
  bool bIsDefault,
  CDXUTButton** ppCreated
)
```

Для вызова метода требуется указать следующие параметры:

- `ID` — идентификатор элемента интерфейса;
- `strText` — текст, который должен отображаться на кнопке;
- `x` и `y` — положение верхнего левого угла элемента относительно осей X и Y;
- `width` и `height` — ширина и высота элемента в пикселах;
- `nHotkey` — код "горячей клавиши", которая позволяет "нажимать" кнопку, используя клавиатуру, значение 0, установленное по умолчанию, значит, что горячая клавиша не определена;
- `bIsDefault` — параметр, который определяет, является ли элемент элементом по умолчанию (т. е. установлен ли на него фокус ввода), если он не задан, то подразумевается значение `FALSE`;
- `ppCreated` — адрес указателя, который может быть установлен методом на создаваемый элемент, по умолчанию установлено значение `NULL`.

Кнопка добавляется так же, как и статичный текст, здесь все понятно. Гораздо больше вопросов вызывает функция `OnGUIEvent()`. Какие параметры ей передает каркас и как организовать ее работу? Рассмотрим прототип этой функции:

```
void CALLBACK OnGUIEvent(
  UINT nEvent,
  int nControlID,
  CDXUTControl* pControl,
  void* pUserContext
);
```

Параметры функции означают следующее:

- `nEvent` — идентификатор события;
- `nControlID` — идентификатор элемента интерфейса, от которого поступило событие;
- `pControl` — указатель на объект базового класса элемента, от которого поступило событие;
- `pUserContext` — указатель на массив данных пользователя.

Какие идентификаторы событий используются в данном случае? Речь идет не о сообщениях операционной системы, эти идентификаторы используются только каркасом DXUT. Возможные значения идентификатора события приведены в табл. 11.1.

Таблица 11.1. *Идентификаторы событий интерфейса*

Идентификатор	Описание
EVENT_BUTTON_CLICKED	Произошел щелчок мышью по кнопке
EVENT_COMBOBOX_SELECTION_CHANGED	Выделен другой элемент в поле со списком
EVENT_RADIOBUTTON_CHANGED	Изменено состояние переключателя
EVENT_CHECKBOX_CHANGED	Изменено состояние флажка
EVENT_SLIDER_VALUE_CHANGED	Изменено значение, установленное с помощью ползунка
EVENT_EDITBOX_STRING	Введена строка с помощью поля ввода (нажата клавиша <Enter>)
EVENT_EDITBOX_CHANGE	Изменен текст, содержащийся в поле ввода
EVENT_LISTBOX_ITEM_DBLCLK	Произошел двойной щелчок мышью по элементу списка
EVENT_LISTBOX_SELECTION	Изменен выделенный пункт в списке

Внутреннее устройство функции `OnGUIEvent()` похоже на функцию `WndProc()`: выбор предпринимаемых действий тоже осуществляется с помощью оператора `switch`, только в качестве параметра выступает идентификатор элемента интерфейса. Как видно из таблицы, для некоторых элементов имеется только одно возможное событие (например, флажок или кнопка), при обработке событий для таких элементов проверять, чему равен идентификатор события, необходимости нет. Однако его обязательно нужно проверять, например, для поля ввода, иначе мы не сможем определить, нужно ли нам принимать введенную пользователем строку, или событие наступило просто потому, что изменился введенный в поле ввода текста. На этом мы еще остановимся. А сейчас посмотрим, как обрабатывать события для нашей единственной кнопки:

```
void CALLBACK OnGUIEvent( UINT nEvent, int nControlID,
                          CDXUTControl* pControl, void* pUserContext )
{
  switch( nControlID )
  {
  case IDC_TOGGLEFULLSCREEN:
    // Переключим режим (оконный/полный экран)
    DXUTToggleFullScreen();
```

```
    // В соответствии с текущим режимом
    // скорректируем текст на кнопке
    if ( DXUTIsWindowed() ) {
      g_UserInterface.GetButton( IDC_TOGGLEFULLSCREEN )->
                              SetText(L"Полноэкранный режим");
    } else
    {
      g_UserInterface.GetButton( IDC_TOGGLEFULLSCREEN )->
                              SetText(L"Оконный режим");
    }

    break;
  }
}
```

Если событие поступило от кнопки переключения режимов, то вызывается функция `DXUTToggleFullScreen()`, которая и производит переключение: если установлен оконный режим, она переключит программу в полноэкранный, и наоборот. После этого, при помощи функции `DXUTIsWindowed()` мы выясняем, в каком режиме находится программа в данный момент: если в оконном, функция возвращает значение `TRUE`. В соответствии с этим значением мы меняем текст на кнопке, для чего используем метод `SetText`. Указатель на кнопку мы получаем с помощью метода `GetButton` диалогового окна.

С обработкой событий мы пока закончили, нам осталось позаботиться о том, чтобы кнопка оставалась в правом верхнем углу при изменении размеров окна. Вы, скорее всего, уже догадываетесь, как это сделать. Открываем функцию `OnD3D10ResizedSwapChain()` и вставляем такую строку:

```
g_UserInterface.GetControl( IDC_TOGGLEFULLSCREEN )->
                    SetLocation( pBackBufferSurfaceDesc->Width-160, 25 );
```

Поставленную задачу мы выполнили, пробуем скомпилировать и запустить проект. В окне программы появилась кнопка с надписью "Полноэкранный режим" (рис. 11.11), щелкнем по ней мышью.

Программа переключится в полноэкранный режим, кнопка переместится в правый верхний угол экрана, а текст на кнопке изменится на "Оконный режим".

Соответствующий проект находится на прилагаемом компакт-диске в директории Glava11\DXUT_Button.

Теперь для того, чтобы добавить новый элемент интерфейса, нужно ввести его инициализацию в функцию `InitApp()`, определить обработку событий

в функции `OnGUIEvent()` и задать размещение в зависимости от размеров окна в функции `OnD3D10ResizedSwapChain()`. Будем иметь в виду эти требования и продолжим знакомство с пользовательским интерфейсом.

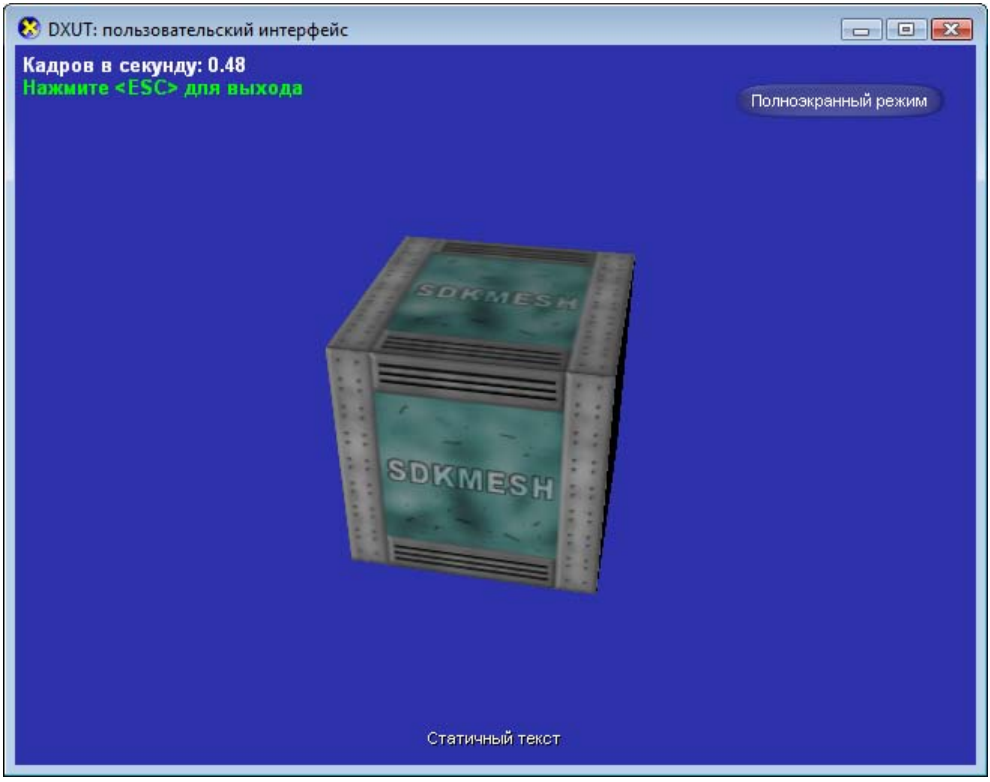

Рис. 11.11. Кнопка переключения в полноэкранный режим и обратно

Добавляем флажок

Элементы интерфейса, которые мы уже использовали, пока не дают нам возможность управлять ходом программы. Наша программа все еще может обойтись без интерфейса вообще: переключение в полноэкранный режим можно осуществлять комбинацией клавиш <Alt>+<Enter>. Давайте добавим флажок, расположенный в нижней части экрана, который контролирует вращение модели вокруг оси Y. Если флажок установлен, модель вращается, если нет — вращения не будет. Также определим для флажка "горячую клавишу".

Введем глобальную переменную, в которой будет храниться состояние флажка:

```
bool      g_bRotate;
```

Зададим идентификатор для нового элемента интерфейса (флажка):

```
#define IDC_TOGGLESPIN          3
```

Переходим к функции инициализации. Флажок добавляется с помощью метода `AddCheckBox()` диалогового окна, при этом функция инициализации примет следующий вид:

```
void InitApp()
{
  g_bRotate = TRUE;
  // Инициализируем интерфейс
  g_UserInterface.Init( &g_DialogResourceManager );

  // Установим функцию для обработки событий
  g_UserInterface.SetCallback( OnGUIEvent );

  g_UserInterface.AddStatic( IDC_STATICTEXT, L"Статичный текст",
                             200, 100, 90, 24 );
  g_UserInterface.AddButton( IDC_TOGGLEFULLSCREEN,
                             L"Полноэкранный режим", 480, 25, 140, 25 );
  g_UserInterface.AddCheckBox( IDC_TOGGLESPIN,
              L"Вращать вокруг оси Y (установка/снятие клавишей <Y>)",
              145, 450, 350, 25, g_bRotate, L'Y');
}
```

Рассмотрим прототип метода, чтобы разобраться с его параметрами:

```
HRESULT AddCheckBox(
  int ID,
  LPCWSTR strText,
  int x,
  int y,
  int width,
  int height,
  bool bChecked,
  UINT nHotkey,
  bool bIsDefault,
  CDXUTCheckBox** ppCreated
)
```

Не будем перечислять все параметры в очередной раз, отметим только параметр `bChecked`, особый параметр для флажка. Он определяет, каким является первоначальное состояние флажка, установлен он или сброшен. Значение `TRUE` — флажок установлен. Обратим также внимание, каким образом устанавливается "горячая клавиша", мы просто передаем соответствующий ей символ.

> ***Замечание***
>
> Чтобы использовать в качестве "горячих клавиш" функциональные клавиши (<F1>..<F12>), нужно указывать их виртуальные коды. Например, для использования клавиши <F1> нужно указать ее код: `VK_F1`.

Теперь, когда мы инициализировали новый элемент, нужно определить, какие действия выполняются при наступлении того или иного события. К счастью, для флажка событие предусмотрено только одно: изменение состояния. Наша задача — изменить значение глобальной переменной `g_bRotate` в соответствии с новым состоянием флажка. Функция `OnGUIEvent()` при этом будет выглядеть таким образом:

```
void CALLBACK OnGUIEvent( UINT nEvent, int nControlID,
                          CDXUTControl* pControl, void* pUserContext )
{
  switch( nControlID )
  {
  case IDC_TOGGLEFULLSCREEN:
    //Переключим режим (оконный/полный экран)
    DXUTToggleFullScreen();
    //В соответствии с текущим режимом
    // скорректируем текст на кнопке
    if ( DXUTIsWindowed() ) {
      g_UserInterface.GetButton( IDC_TOGGLEFULLSCREEN )->
                          SetText(L"Полноэкранный режим");
    } else
    {
      g_UserInterface.GetButton( IDC_TOGGLEFULLSCREEN )->
                          SetText(L"Оконный режим");
    }
    break;

  case IDC_TOGGLESPIN:
    {
```

```
      g_bRotate = g_UserInterface.GetCheckBox( IDC_TOGGLESPIN )->
                                                    GetChecked();
      break;
    }
  }
}
```

В случае если событие поступило от флажка, мы используем метод флажка `GetChecked()` для получения информации о его состоянии и полученное значение присваиваем переменной `g_bRotate`. Указатель на объект флажка мы получаем при помощи метода `GetCheckBox()` диалогового окна. Пока что мы запрограммировали только изменение глобальной переменной в зависимости от состояния флажка, на вращение модели это еще никак не влияет. Давайте изменим функцию `OnFrameMove()`, в которой формируется мировая матрица. Новая версия функции будет выглядеть следующим образом:

```
void CALLBACK OnFrameMove( double fTime, float fElapsedTime,
                           void* pUserContext )
{
  // Обновляем положение камеры
  g_Camera.FrameMove(fElapsedTime);

  // Получаем параметры устройства Direct3D
  DXUTDeviceSettings D3D10DevSet;
  D3D10DevSet = DXUTGetDeviceSettings();
  static float t = 0.0f;

  // Если включено вращение вокруг оси Y...
  if ( g_bRotate )
  {
    // Устанавливаем угол поворота
    // в зависимости от типа драйвера устройства
    if (D3D10DevSet.d3d10.DriverType == D3D10_DRIVER_TYPE_REFERENCE)
      t += (float)D3DX_PI * 0.0125f;
    else
      t += fElapsedTime;
  }

  // Заполняем матрицу вращения
  D3DXMATRIX mRotY;
  D3DXMatrixRotationY( &mRotY, t );
```

```
  // Устанавливаем мировую матрицу
  g_World = *g_Camera.GetWorldMatrix();
  g_World = g_World * mRotY;
}
```

В этой функции угол вращения увеличивается только в том случае, если установлен флажок или, что то же самое, переменная `g_bRotate` содержит значение `TRUE`. Значение угла используется для формирования матрицы вращения, которая потом объединяется путем перемножения с мировой матрицей, полученной от камеры.

Основную часть работы мы выполнили, теперь переключимся на функцию `OnD3D10ResizedSwapChain()` и сделаем так, чтобы флажок все время находился посередине нижней части экрана, а также скорректируем положение статичного текста таким образом, чтобы он располагался над флажком. Вот так будет выглядеть функция после внесения изменений:

```
HRESULT CALLBACK OnD3D10ResizedSwapChain( ID3D10Device* pd3dDevice,
                        IDXGISwapChain *pSwapChain,
                        const DXGI_SURFACE_DESC* pBackBufferSurfaceDesc,
                        void* pUserContext )
{
  HRESULT hr;

  V_RETURN( g_DialogResourceManager.OnD3D10ResizedSwapChain(
pd3dDevice, pBackBufferSurfaceDesc ) );

  // Устанавливаем параметры матрицы проекции
  float fAspect = (float)( pBackBufferSurfaceDesc->Width )/
                              (float)( pBackBufferSurfaceDesc->Height );
  g_Camera.SetProjParams( D3DX_PI * 0.25f, fAspect, 0.1f, 100.0f );
  g_Camera.SetWindow( pBackBufferSurfaceDesc->Width,
                      pBackBufferSurfaceDesc->Height );
  g_Camera.SetButtonMasks( MOUSE_LEFT_BUTTON,
                           MOUSE_WHEEL, MOUSE_RIGHT_BUTTON );

  // Передвигаем элементы интерфейса, подстраиваемся под новые
  // размеры окна
  // Размещаем элемент посередине окна:
  // берем ширину элемента
  int StaticWidth = g_UserInterface.GetControl( IDC_STATICTEXT )->
                                                        m_width;
```

```
  int CheckBoxWidth = g_UserInterface.GetControl( IDC_TOGGLESPIN )->
                                                        m_width;
  int WindowWidth   = pBackBufferSurfaceDesc->Width;
  // Зная ширину элемента и ширину окна,
  // рассчитываем координату X : (WindowWidth-StaticWidth)/2
  g_UserInterface.GetControl( IDC_STATICTEXT )->SetLocation(
       (WindowWidth-StaticWidth)/2, pBackBufferSurfaceDesc->Height-50);
  g_UserInterface.GetControl( IDC_TOGGLESPIN )->SetLocation(
       (WindowWidth-CheckBoxWidth)/2, pBackBufferSurfaceDesc->Height-30);

  g_UserInterface.GetControl( IDC_TOGGLEFULLSCREEN )->SetLocation(
       pBackBufferSurfaceDesc->Width-160, 25 );

  return S_OK;
}
```

Теперь мы можем запустить программу и попробовать ее в работе. Как и следовало ожидать, мы увидим новый элемент интерфейса — флажок (рис. 11.12).

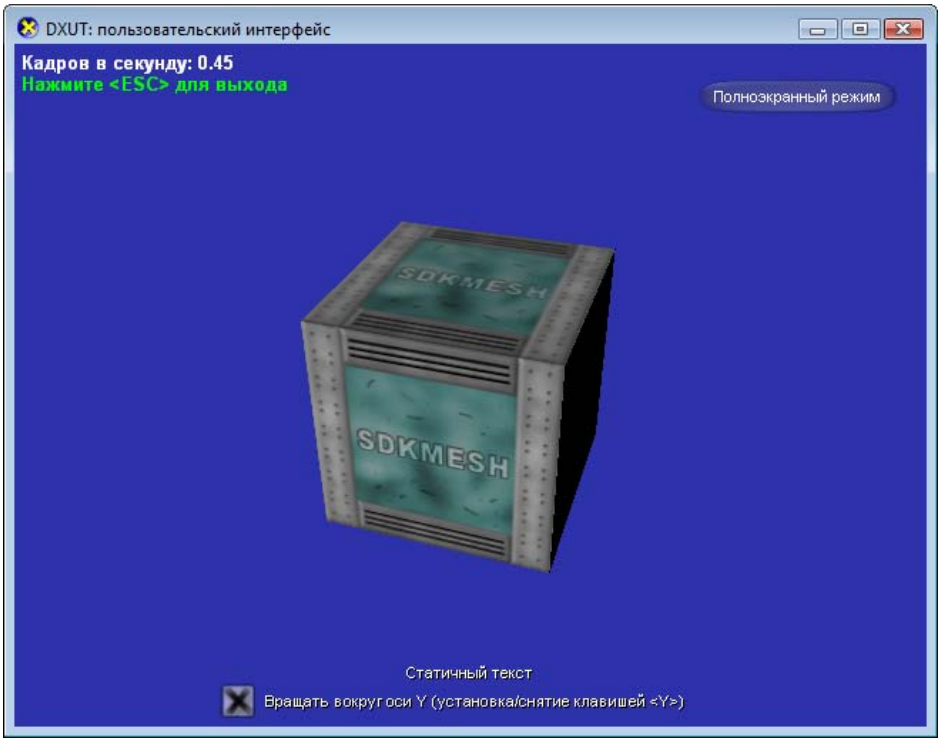

Рис. 11.12. Флажок для управления вращением модели

Щелкнем по нему мышью. Мы сняли с флажка знак отметки, и вращение модели прекратилось. Щелкнем по флажку еще раз, и модель снова придет в движение.

Готовый проект, в котором реализовано управление вращением модели, находится на компакт-диске в директории Glava11\DXUT_CheckBox.

Добавляем поле ввода

Попытаемся добавить в программу поле ввода. Расположим его также в нижней части окна, освободим для него место, сдвинув статичный текст немного вверх. Статичный текст, кстати говоря, тоже пусть не простаивает: его можно использовать для отражения изменений, происходящих с текстом в поле ввода.

Ввод нового элемента интерфейса начинаем с установки его идентификатора:

```
#define IDC_EDITBOX        4
```

Поле этого переходим в функцию инициализации, где при помощи метода `AddEditBox()` диалогового окна добавляется поле ввода. Метод имеет те же самые параметры, что и метод `AddStatic`. Вот как будет выглядеть функция инициализации `InitApp()`:

```
void InitApp()
{
  g_bRotate = TRUE;
  // Инициализируем интерфейс
  g_UserInterface.Init( &g_DialogResourceManager );

  // Установим функцию для обработки событий
  g_UserInterface.SetCallback( OnGUIEvent );

  g_UserInterface.AddStatic(IDC_STATICTEXT,
            L"Пример\n статичного текста", 200, 430, 640, 25);
  g_UserInterface.AddButton( IDC_TOGGLEFULLSCREEN,
            L"Полноэкранный режим", 480, 25, 140, 25 );
  g_UserInterface.AddCheckBox( IDC_TOGGLESPIN,
            L"Вращать вокруг оси Y (установка/снятие клавишей <Y>)",
            145, 450, 350, 25, g_bRotate, L'Y');
  g_UserInterface.AddEditBox( IDC_EDITBOX,
            L"Введите текст и нажмите <Enter>", 200, 420, 250, 32);
}
```

Обратите внимание, что немного изменилась форма подачи статичного текста: теперь он выводится в две строчки: мы ввели в текст символ перевода

строки \n. Воспользуемся этой более удобной формой для вывода на экран введенного текста. Отредактируем функцию обработки событий:

```
void CALLBACK OnGUIEvent( UINT nEvent, int nControlID,
                          CDXUTControl* pControl, void* pUserContext )
{
  //Буфер для передачи текста из окна ввода
  WCHAR wszOutput[1024];

  switch( nControlID )
  {
  case IDC_TOGGLEFULLSCREEN:
    //Переключим режим (оконный/полный экран)
    DXUTToggleFullScreen();

    //В соответствии с текущим режимом
    // скорректируем текст на кнопке
    if ( DXUTIsWindowed() ) {
      g_UserInterface.GetButton( IDC_TOGGLEFULLSCREEN )->
                            SetText(L"Полноэкранный режим");
    } else
    {
      g_UserInterface.GetButton( IDC_TOGGLEFULLSCREEN )->
                            SetText(L"Оконный режим");
    }
    break;

  case IDC_TOGGLESPIN:
    {
      g_bRotate = g_UserInterface.GetCheckBox( IDC_TOGGLESPIN )->
                                                    GetChecked();
    }
    break;

  case IDC_EDITBOX:
     {
       switch( nEvent )
       case EVENT_EDITBOX_STRING:
       {
```

```
            StringCchPrintf( wszOutput, 1024, L"Введена строка \n\"%s\"",
                             ((CDXUTEditBox*)pControl)->GetText());

            g_UserInterface.GetStatic( IDC_STATICTEXT )->
                                                SetText( wszOutput );
            break;
          }
        }
        break;
    }
}
```

В этой функции реализуется отображение введенного в поле ввода текста с помощью другого элемента интерфейса. Обратите внимание, что здесь присутствует еще один оператор `switch`, так как от поля ввода может приходить два вида событий, из них мы обрабатываем только `EVENT_EDITBOX_STRING`.

Сначала мы с помощью функции `StringCchPrintf()` формируем строку символов, которую нужно вывести как статичный текст. Эта функция является заменой для функции `sprintf()`. Разберем прототип функции `StringCchPrintf()` и все, что мы используем в качестве параметров:

```
HRESULT StringCchPrintf(
    LPTSTR pszDest,
    size_t cchDest,
    LPCTSTR pszFormat,
     ...
);
```

Функция требует указания следующих параметров:

- `pszDest` — указатель на целевой буфер, в который помещается итоговая отформатированная строка символов;
- `cchDest` — размер целевого буфера, выраженный в символах (буфер должен быть достаточно большим, чтобы вместить все символы итоговой строки плюс нулевой символ завершения);
- `pszFormat` — указатель на буфер, содержащий строку с символами форматирования, строка должна завершаться нулевым символом;
- `...` — список аргументов, которые нужно вставить в итоговую строку символов.

В качестве буфера мы используем массив `wszOutput`, в котором помещается 1024 символа. Строка форматирования содержит только один "маркер", он указывает на то, что в этом месте необходимо вставить строку символов пер-

вого параметра из списка. Первым параметром у нас идет строка символов, извлеченная из поля ввода с помощью метода `GetText()`. Этот метод мы вызвали через указатель на базовый объект, который привели к указателю на экземпляр класса `CDXUTEditBox`. Итоговую строку символов из массива `wszOutput` мы передаем в элемент статичного текста, используя уже знакомый нам метод `SetText()`.

Сохраним положение поля ввода посередине окна при изменении его размеров. Поступаем аналогично тому, как мы вышли из положения со статичным текстом и флажком. В этом случае функция `OnD3D10ResizedSwapChain()` приобретет следующий вид:

```
HRESULT CALLBACK OnD3D10ResizedSwapChain( ID3D10Device* pd3dDevice,
                          IDXGISwapChain *pSwapChain,
                          const DXGI_SURFACE_DESC* pBackBufferSurfaceDesc,
                          void* pUserContext )
{
  HRESULT hr;

  V_RETURN( g_DialogResourceManager.OnD3D10ResizedSwapChain(
                               pd3dDevice, pBackBufferSurfaceDesc ) );

  // Устанавливаем параметры матрицы проекции
  float fAspect = (float)( pBackBufferSurfaceDesc->Width )/
                              (float)( pBackBufferSurfaceDesc->Height );
  g_Camera.SetProjParams( D3DX_PI * 0.25f, fAspect, 0.1f, 100.0f );
  g_Camera.SetWindow( pBackBufferSurfaceDesc->Width,
                      pBackBufferSurfaceDesc->Height );
  g_Camera.SetButtonMasks( MOUSE_LEFT_BUTTON, MOUSE_WHEEL,
                           MOUSE_RIGHT_BUTTON );

  // Передвигаем элементы интерфейса, подстраиваемся под новые
  // размеры окна
  // Размещаем элемент посередине окна:
  // берем ширину нашего элемента
  int CheckBoxWidth =
               g_UserInterface.GetControl( IDC_TOGGLESPIN )->m_width;
  int EditBoxWidth =
               g_UserInterface.GetControl( IDC_EDITBOX )->m_width;
  int WindowWidth = pBackBufferSurfaceDesc->Width;
```

```
    // Зная ширину элемента и ширину окна
    // рассчитываем координату X : (WindowWidth-StaticWidth)/2
    g_UserInterface.GetControl( IDC_TOGGLESPIN )->SetLocation(
       (WindowWidth-CheckBoxWidth)/2 , pBackBufferSurfaceDesc->Height-30 );
    g_UserInterface.GetControl( IDC_EDITBOX )->SetLocation (
       (WindowWidth-EditBoxWidth)/2, pBackBufferSurfaceDesc->Height-60 );

    // У статичного текста нужно менять и координаты положения, и
    // размеры
    g_UserInterface.GetControl( IDC_STATICTEXT )->SetLocation( 0,
                                    pBackBufferSurfaceDesc->Height-90 );
    g_UserInterface.GetControl( IDC_STATICTEXT )->
                                            SetSize( WindowWidth, 32 );

    g_UserInterface.GetControl( IDC_TOGGLEFULLSCREEN )->SetLocation(
                                    pBackBufferSurfaceDesc->Width-160, 25 );

    return S_OK;
}
```

Если вы внимательно изучили текст функции, то наверняка заметили одну особенность, относящуюся к статичному тексту. Она связана с тем, что количество символов, которое будет вмещать элемент по ширине, нам заранее неизвестно, а значит, при инициализации мы не можем установить соответствующую ширину элемента. Как быть? Мы поступаем следующим образом: устанавливаем ширину статичного текста во всю ширину вторичного буфера. Строка символов, отображаемая статичным текстом, автоматически центрируется по ширине элемента, и таким образом мы получаем отображение по центру практически любого количества символов (точнее, такого количества, которое помещается в буфере — 1023 символа). В связи с этим, при изменении размеров окна нам нужно менять как положение статичного текста, "прижимая" его к левому краю, так и размер элемента по горизонтали, подгоняя его под новую ширину окна.

Компилируем проект, запускаем его, чтобы проверить в работе. К элементам интерфейса программы добавилось поле ввода (рис. 11.13).

Вводим в поле ввода несколько символов. Все символы, содержащиеся в поле ввода, немедленно отображаются статичным текстом (рис. 11.14).

Готовый проект с добавленным полем ввода можно найти на компакт-диске в директории Glava11\DXUT_EditBox.

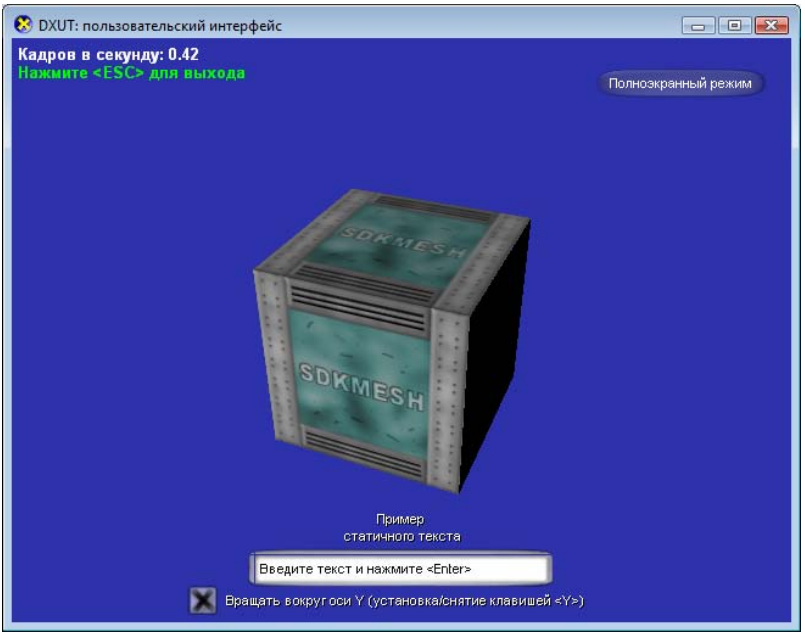

Рис. 11.13. Окно программы с полем ввода

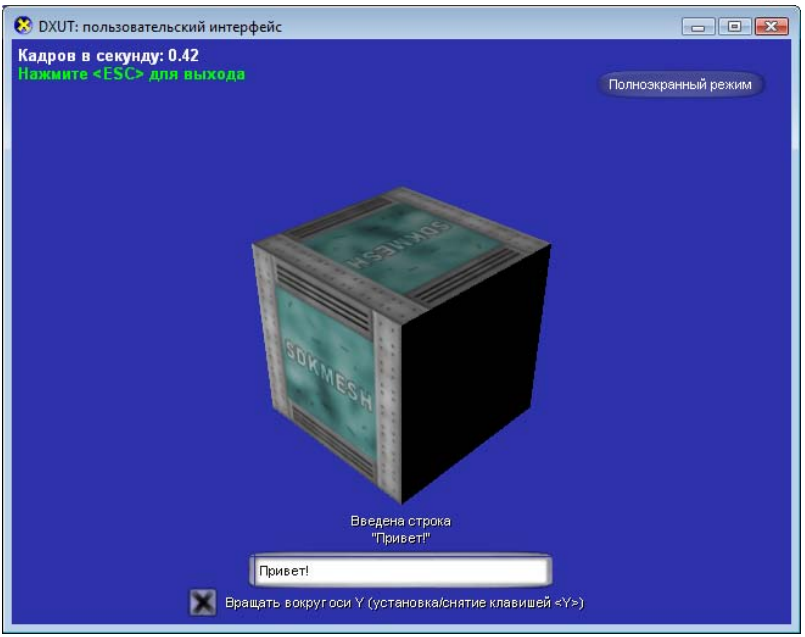

Рис. 11.14. Вид окна программы после изменения текста в поле ввода

Добавляем выпадающий список

Пополним нашу коллекцию элементов интерфейса и добавим в нашу программу выпадающий список. Давайте разместим этот элемент под кнопкой переключения программы в полноэкранный режим и обратно. Пусть в выпадающем списке будет три элемента. При изменении выделенного элемента в выпадающем списке будем выводить сообщения через статичный текст.

Начнем мы, как обычно, с того, что зададим идентификатор для нового элемента:

```
#define IDC_COMBOBOX        5
```

После этого дополним функцию инициализации интерфейса. Инициализация выпадающего списка несколько отличается от того, что нам встречалось раньше:

```
void InitApp()
{
  g_bRotate = TRUE;
  // Инициализируем интерфейс
  g_UserInterface.Init( &g_DialogResourceManager );

  // Установим функцию для обработки событий
  g_UserInterface.SetCallback( OnGUIEvent );

  g_UserInterface.AddStatic( IDC_STATICTEXT,
                L"Пример\n статичного текста", 200, 430, 640, 25);
  g_UserInterface.AddButton( IDC_TOGGLEFULLSCREEN,
                L"Полноэкранный режим", 480, 25, 140, 25 );
  g_UserInterface.AddCheckBox( IDC_TOGGLESPIN,
                L"Вращать вокруг оси Y (установка/снятие клавишей <Y>)",
                145, 450, 350, 25, g_bRotate, L'Y' );
  g_UserInterface.AddEditBox( IDC_EDITBOX,
                L"Введите текст и нажмите <Enter>", 200, 420, 250, 32);

  // Добавляем и заполняем выпадающий список (ComboBox)
  CDXUTComboBox *pCombo;
  g_UserInterface.AddComboBox( IDC_COMBOBOX, 480, 55, 140, 25,
                               L'C', false, &pCombo );
  if( pCombo )
  {
    pCombo->SetDropHeight( 40 );
```

```
    pCombo->AddItem( L"Элемент 1", (LPVOID)0x11111111 );
    pCombo->AddItem( L"Элемент 2", (LPVOID)0x22222222 );
    pCombo->AddItem( L"Элемент 3", (LPVOID)0x33333333 );
  }
}
```

Выпадающий список добавляется к элементам интерфейса при помощи метода `AddComboBox()` диалогового окна. Прототип уже, наверное, рассматривать не нужно, все ясно при взгляде на передаваемые значения. Явно видно, что передаются координаты положения элемента, его ширина и высота в пикселах, код "горячей клавиши", значение, указывающее, является ли список элементом по умолчанию, а также адрес указателя, который будет установлен на созданный элемент. Здесь мы как раз используем указатель на созданный элемент для последующего заполнения выпадающего списка.

После того как элемент создан, мы сначала вызываем его метод `SetDropHeight()` и устанавливаем высоту выпадающего списка равной сорока пикселам. Далее мы используем метод `AddItem()` для добавления элементов списка. В параметрах мы передаем текст, которым элемент обозначается в списке, и указатель на данные, связанные с этим элементом.

Позаботимся об обработке событий, поступающих от поля со списком. В функцию `OnGUIEvent()` при этом нужно добавить следующую ветвь для оператора `switch`:

```
case IDC_COMBOBOX:
  {
    DXUTComboBoxItem *pItem =
                      ((CDXUTComboBox*)pControl)->GetSelectedItem();

    if( pItem )
    {
      StringCchPrintf( wszOutput, 1024,
          L"Выбран новый элемент:\n \"%s\", с ним связано значение 0x%p",
          pItem->strText, pItem->pData );
      g_UserInterface.GetStatic( IDC_STATICTEXT )-> SetText( wszOutput );
    }
  }
  break;
```

Здесь мы сначала с помощью метода `GetSelectedItem()` получаем указатель на выбранный элемент списка. После этого при помощи функции `StringCchPrintf()` форматируем строку, которая содержит текст элемента и связанное с ним значение. Итоговую строку передаем в статичный текст.

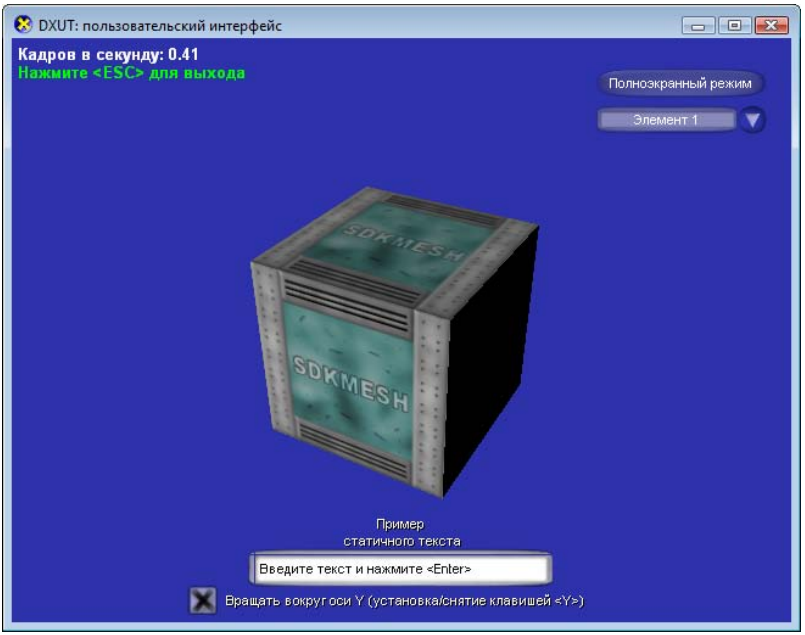

Рис. 11.15. Окно программы с добавленным выпадающим списком

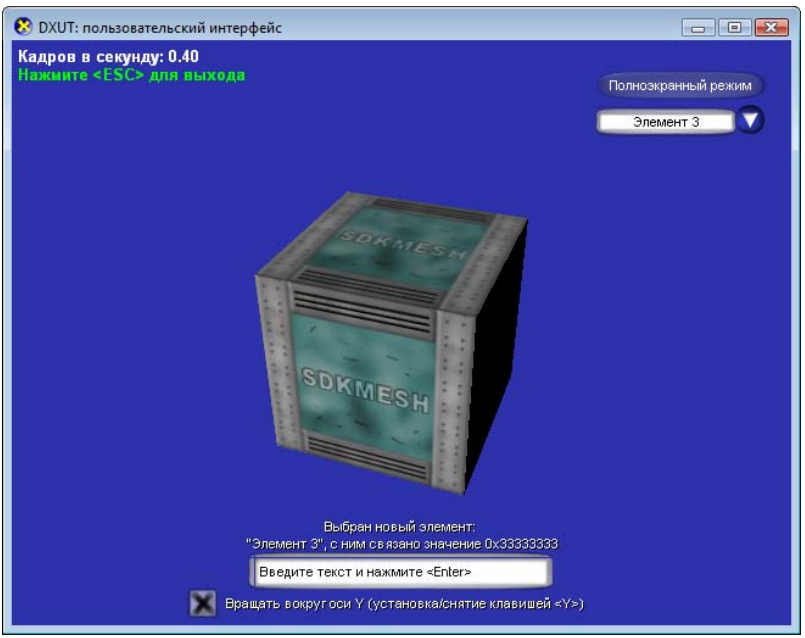

Рис. 11.16. Отображение выделенного элемента выпадающего списка

При изменении размеров окна нужно проследить, чтобы выпадающий список оставался под кнопкой, для этого добавим в функцию `OnD3D10ResizedSwapChain()` строку:

```
g_UserInterface.GetControl( IDC_COMBOBOX )->
                SetLocation( pBackBufferSurfaceDesc->Width-160, 55 );
```

Теперь мы можем запустить проект и полюбоваться результатами своих усилий. Под кнопкой переключения экранных режимов мы увидим новый элемент: выпадающий список (рис. 11.15).

Щелкнем по нему мышью и выберем какой-нибудь элемент в выпадающем списке. Сразу после нашего выбора текст элемента отобразится в элементе статичного текста (рис. 11.16).

Нажмем на клавишу <C> и увидим переключение выделенного элемента выпадающего списка после каждого нажатия.

Соответствующий проект находится на компакт-диске в директории Glava11\DXUT_ComboBox.

Добавляем список

Действия, которые необходимо выполнить для использования списка в программе, во многом схожи с теми, что мы выполняли при добавлении выпадающего списка. Давайте добавим список к элементам интерфейса программы, разместим мы его под выпадающим списком и будем обрабатывать два события: выделение пункта списка и двойной щелчок мышью по элементу списка. Сообщения о выделенных элементах будем выводить при помощи того же самого элемента статичного текста.

Определяем идентификатор нового элемента:

```
#define IDC_LISTBOX    6
```

Инициализацию и заполнение списка проведем немного иначе, чем в предыдущем случае:

```
  // Добавляем список (ListBox) и его элементы
  g_UserInterface.AddListBox( IDC_LISTBOX, 480, 85, 140, 70, 0 );
  g_UserInterface.GetListBox( IDC_LISTBOX )->AddItem( L"Вариант 1",
                                                      (LPVOID)0x11112222 );
  g_UserInterface.GetListBox( IDC_LISTBOX )->AddItem( L"Вариант 2",
                                                      (LPVOID)0x22222222 );
  g_UserInterface.GetListBox( IDC_LISTBOX )->AddItem( L"Вариант 3",
                                                      (LPVOID)0x33332222 );
```

```
  g_UserInterface.GetListBox( IDC_LISTBOX )->AddItem( L"Вариант 4",
                                                      (LPVOID)0x44442222 );
  g_UserInterface.GetListBox( IDC_LISTBOX )->AddItem( L"Вариант 5",
                                                      (LPVOID)0x55552222 );
```

Добавляется список методом `AddListBox()` диалогового окна. Первыми передаются параметры, определяющие положение и размеры элемента, самым последним идет параметр, задающий стиль списка. В нашем случае передается нулевое значение, это значит, что в списке допускается только одиночное выделение элементов. Если нужно использовать список с возможностью одновременного выделения нескольких элементов, необходимо указать значение `CDXUTListBox::MULTISELECTION`. Заполнение списка производится без использования параметра, в который помещается указатель на созданный элемент. При добавлении каждого пункта списка мы получаем этот указатель, используя метод `GetListBox` диалогового окна.

Чтобы реализовать реакцию на выделение нового элемента и на двойной щелчок, нам нужно добавить в функцию `OnGUIEvent()` следующие строки:

```
case IDC_LISTBOX:
  switch( nEvent )
  {
   case EVENT_LISTBOX_SELECTION:
   {
     StringCchPrintf( wszOutput, 1024,
           L"Выбран новый элемент в списке,\n это элемент с индексом %d",
           ((CDXUTListBox *)pControl)->GetSelectedIndex() );
     g_UserInterface.GetStatic( IDC_STATICTEXT )->SetText( wszOutput );
     break;
   }

   case EVENT_LISTBOX_ITEM_DBLCLK:
   {
     DXUTListBoxItem *pItem = ((CDXUTListBox *)pControl)->
           GetItem( ((CDXUTListBox *)pControl)->GetSelectedIndex( -1 ) );

     StringCchPrintf( wszOutput, 1024,
              L"Двойной щелчок по элементу списка,\n это элемент \"%s\"",
              pItem ? pItem->strText : L"" );
     g_UserInterface.GetStatic( IDC_STATICTEXT )->SetText( wszOutput );
     break;
   }
  }
  break;
```

Здесь нужно пояснить, что индекс выделенного элемента мы получаем с помощью метода `CDXUTListBox::GetSelectedIndex()`, значение параметра `-1` указывает, что будет возвращен указатель на первый найденный выделенный элемент. Это нас вполне устраивает, так как в нашем случае выделенных элементов больше одного быть не может.

В случае, когда обрабатывается двойной щелчок по элементу списка, мы получаем указатель на элемент списка при помощи метода `CDXUTListBox::GetItem()`. Этот метод возвращает указатель на тот элемент списка, индекс которого передан ему через параметр. Чтобы получить индекс выделенного элемента, мы снова воспользуемся методом `CDXUTListBox::GetSelectedIndex()`. Таким образом, в результате у нас получился указатель на выделенный элемент. Для получения текста элемента мы используем поле `strText` структуры типа `DXUTListBoxItem`, на которую указывает полученный указатель.

Чтобы сохранить положение списка относительно других элементов, давайте внесем в функцию `OnD3D10ResizedSwapChain()` еще одну строку:

```
g_UserInterface.GetControl( IDC_LISTBOX )->
                    SetLocation( pBackBufferSurfaceDesc->Width-160, 85 );
```

Скомпилируем проект и проверим его работоспособность. Общий вид окна запущенной программы показан на рис. 11.17.

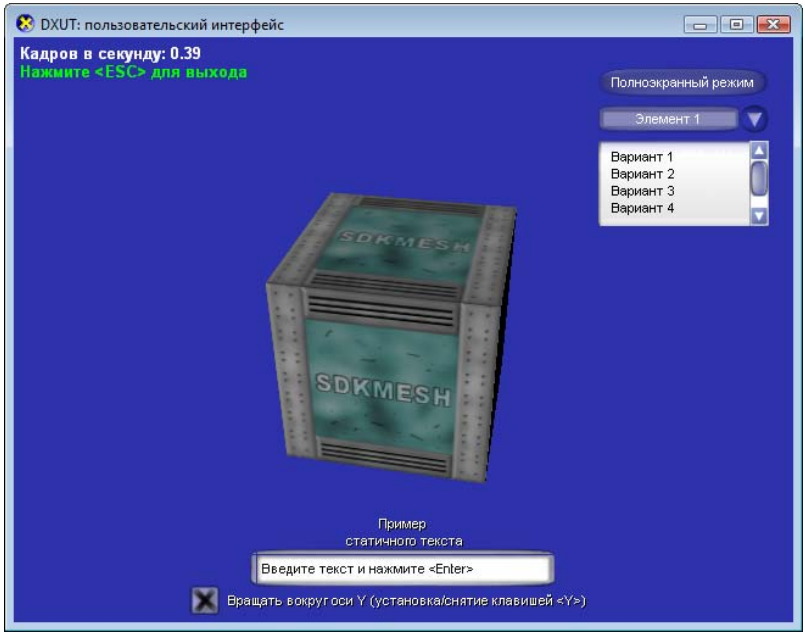

Рис. 11.17. Окно программы с добавленным списком

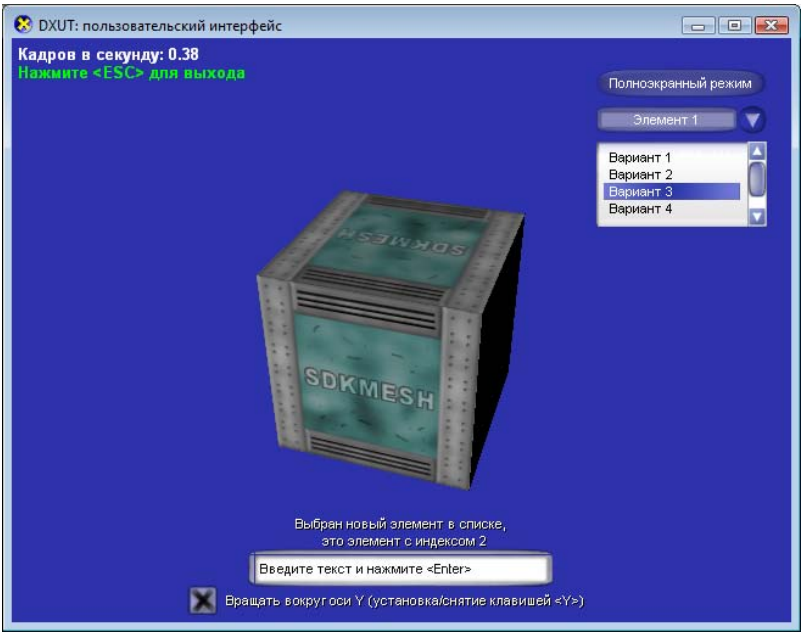

Рис. 11.18. Выделение элемента списка

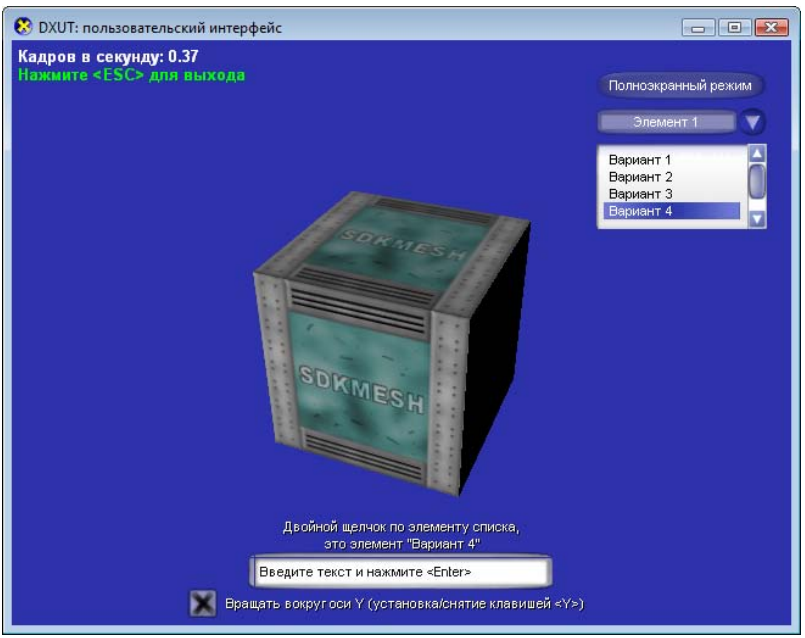

Рис. 11.19. Двойной щелчок по элементу списка

Попробуем выделять разные элементы списка, сразу же будет отображаться текст с указанием индекса вновь выделенного элемента (рис. 11.18).

В завершение наших испытаний проверим, как программа реагирует на двойной щелчок по элементу списка. При двойном щелчке будет выведено тестовое сообщение, содержащее текст выбранного элемента списка (рис. 11.19).

Проект, реализующий добавление к элементам интерфейса списка, находится на прилагаемом компакт-диске в директории Glava11\DXUT_ListBox.

Добавляем ползунок

Добавим в программу ползунок и еще один элемент статичного текста для вывода значений, которые заданы с его помощью. Ползунок и связанный с ним статичный текст расположим в окне программы немного ниже списка.

По сложившейся традиции начнем с добавления определений идентификаторов новых элементов:

```
#define IDC_SLIDER       7
#define IDC_SLIDERTEXT   8
```

Теперь дело за инициализацией, открываем функцию `InitApp()` и вписываем такой текст:

```
// Добавляем ползунок (Slider)
g_UserInterface.AddSlider( IDC_SLIDER, 480, 170, 140, 25,
                           0, 100, 50, false );
// Добавляем статичный текст для отображения значения ползунка
g_UserInterface.AddStatic( IDC_SLIDERTEXT, L"50", 480, 190, 140, 25 );
```

Инициализацию статичного текста мы уже проходили, а вот метод диалогового окна `AddSlider`, с помощью которого мы добавляем ползунок, нужно рассмотреть подробно. Прототип этого метода выглядит следующим образом:

```
HRESULT AddSlider(
  int ID,
  int x,
  int y,
  int width,
  int height,
  int min,
  int max,
  int value,
```

```
  bool bIsDefault,
  CDXUTSlider** ppCreated
)
```

Методу требуется передать такие параметры:

- `ID` — идентификатор элемента интерфейса, закрепляемый за ползунком;
- `x` и `y` — положение верхнего левого угла ползунка относительно осей X и Y;
- `width` и `height` — ширина и высота ползунка в пикселах;
- `min` и `max` — минимальное и максимальное значения, которое можно установить с помощью ползунка;
- `value` — значение ползунка при инициализации;
- `bIsDefault` — флаг, определяющий, является ползунок элементом по умолчанию (установлен ли сразу на него фокус ввода), значение по умолчанию — `FALSE`;
- `ppCreated` — адрес указателя, которому, если он указан, будет присвоено значение, указывающее на создаваемый элемент, по умолчанию установлено значение `NULL`.

Рассмотрев прототип и все параметры, мы теперь можем сказать, что при помощи введенного нами фрагмента текста программы мы инициализировали ползунок с пределом значений от 0 до 100, а при инициализации установлено значение 50.

Обеспечим отображение текущего установленного значения с помощью нового элемента статичного текста:

```
case IDC_SLIDER:
    StringCchPrintf( wszOutput, 1024, L"%d",
                     ((CDXUTSlider*)pControl)->GetValue() );
    g_UserInterface.GetStatic( IDC_SLIDERTEXT )->SetText( wszOutput );
    break;
```

При помощи метода `GetValue()` мы получаем установленное значение и при помощи функции `StringCchPrintf()` форматируем результирующую строку, содержащую это число в виде символов. Строку передаем элементу статичного текста с помощью уже известного нам метода `SetText`.

Нам осталось обеспечить соответствующее положение новых элементов при изменении размеров окна. Установим их положение в функции `OnD3D10ResizedSwapChain()`:

```
g_UserInterface.GetControl( IDC_SLIDER )->SetLocation(
                                  pBackBufferSurfaceDesc->Width-160, 170 );
```

```
g_UserInterface.GetControl( IDC_SLIDERTEXT )->SetLocation(
                                    pBackBufferSurfaceDesc->Width-160, 190 );
```

Теперь можно скомпилировать проект и оценить результат (рис. 11.20).

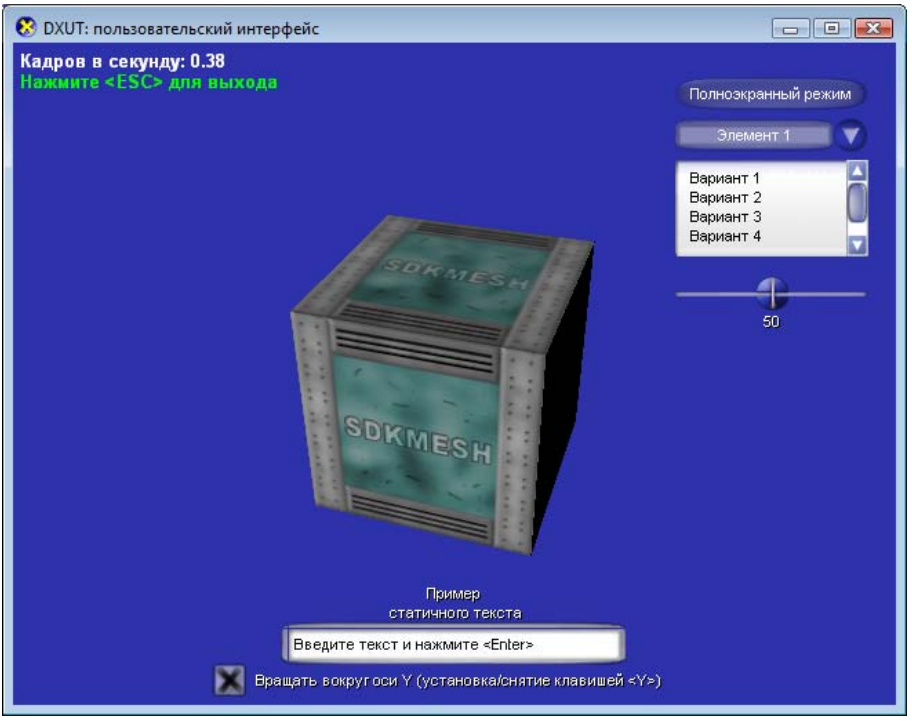

Рис. 11.20. Окно приложения с добавленным ползунком

Переместим движок ползунка при помощи мыши взад и вперед. Посмотрим, как в соответствии с его положением меняется отображаемое значение.

Готовый проект с добавленным ползунком можно найти на компакт-диске в директории Glava11\DXUT_Slider.

Добавляем переключатели

Нерассмотренными у нас остались только переключатели. Давайте для наработки опыта работы с интерфейсом тоже добавим их в программу. Мы добавим три кнопки переключателя, а также статичный текст, который будем использовать для вывода сообщения о том, какая из кнопок в данный момент включена. Заодно установим для кнопок-переключателей "горячие клавиши": <1>, <2> и <3>.

Начинаем с идентификаторов добавляемых элементов интерфейса:

```
#define IDC_RADIOTEXT    9
#define IDC_RADIO1       10
#define IDC_RADIO2       11
#define IDC_RADIO3       12
```

Переходим к инициализации, которая выглядит следующим образом:

```
// Добавляем переключатель и статичный текст к нему
g_UserInterface.AddStatic( IDC_RADIOTEXT, L"Включена\nВЕРХНЯЯ КНОПКА",
                           480, 280, 140, 32);

g_UserInterface.AddRadioButton( IDC_RADIO1, 1, L"Верхняя кнопка <1>",
                                480, 310, 140, 24, true,  L'1' );
g_UserInterface.AddRadioButton( IDC_RADIO2, 1, L"Средняя кнопка <2>",
                                480, 335, 140, 24, false, L'2' );
g_UserInterface.AddRadioButton( IDC_RADIO3, 1, L"Нижняя кнопка  <3>",
                                480, 360, 140, 24, false, L'3' );
```

Инициализация статичного текста для нас уже интереса не представляет. Подробно разберем метод `AddRadioButton()` диалогового окна, его прототип и параметры:

```
HRESULT AddRadioButton(
  int ID,
  UINT nButtonGroup,
  LPCWSTR strText,
  int x,
  int y,
  int width,
  int height,
  bool bChecked,
  UINT nHotkey,
  bool bIsDefault,
  CDXUTRadioButton** ppCreated
)
```

Назначение параметров следующее:

- `ID` — идентификатор элемента интерфейса, закрепляемый за кнопкой переключателя;
- `nButtonGroup` — номер группы кнопок, к которой принадлежит создаваемая кнопка переключателя;
- `strText` — текст, который будет отображаться кнопкой переключателя;

- `x` и `y` — положение верхнего левого угла кнопки переключателя относительно осей X и Y;
- `width` и `height` — ширина и высота кнопки переключателя в пикселах;
- `bChecked` — флаг, показывающий, является ли создаваемая кнопка переключателя выделенной;
- `nHotkey` — код "горячей клавиши", связанной с создаваемой кнопкой переключателя;
- `bIsDefault` — параметр, определяющий, является ли кнопка переключателя элементом по умолчанию (установлен ли сразу на нее фокус ввода), если он не указан, то подразумевается значение `FALSE`;
- `ppCreated` — адрес указателя, который, если он задан, будет указывать на вновь созданный элемент, значение по умолчанию — `NULL`.

Все параметры, кроме одного, мы уже встречали, когда рассматривали другие элементы интерфейса. Параметр, которого нигде больше не было — это номер группы кнопок переключателя. Среди кнопок переключателей с одинаковым номером группы одновременно может быть выделена только одна кнопка. То есть, если мы хотим, например, чтобы у нас на экране было два переключателя, с помощью одного мы выбираем количество источников освещения, а с помощью второго — экранный режим работы программы, нам нужно как-то связать между собой соответствующие кнопки. Вот здесь и нужны группы, чтобы объединить кнопки по смыслу, согласно выполняемым функциям: каждой логической группе переключателей задается свой номер группы.

Реализация смены статичного текста в зависимости от того, какая кнопка переключателя выделена, крайне проста. Мы добавляем ветви оператора `switch` и с помощью метода элемента статичного текста `SetText` устанавливаем нужное сообщение при выделении соответствующей кнопки:

```
case IDC_RADIO1:
  g_UserInterface.GetStatic( IDC_RADIOTEXT )->
                                  SetText(L"Включена\nВЕРХНЯЯ КНОПКА");
  break;

case IDC_RADIO2:
  g_UserInterface.GetStatic( IDC_RADIOTEXT )->
                                  SetText(L"Включена\nСРЕДНЯЯ КНОПКА");
  break;

case IDC_RADIO3:
  g_UserInterface.GetStatic( IDC_RADIOTEXT )->
                                  SetText(L"Включена\nНИЖНЯЯ КНОПКА");
  break;
```

Обеспечиваем правильное положение кнопок переключателя и статичного текста при изменении размеров окна:

```
g_UserInterface.GetControl( IDC_RADIOTEXT )->
                SetLocation( pBackBufferSurfaceDesc->Width-160,
                             pBackBufferSurfaceDesc->Height-200 );

g_UserInterface.GetControl( IDC_RADIO1 )->
                SetLocation( pBackBufferSurfaceDesc->Width-160,
                             pBackBufferSurfaceDesc->Height-170 );

g_UserInterface.GetControl( IDC_RADIO2 )->
                SetLocation( pBackBufferSurfaceDesc->Width-160,
                             pBackBufferSurfaceDesc->Height-145 );

g_UserInterface.GetControl( IDC_RADIO3 )->
                SetLocation( pBackBufferSurfaceDesc->Width-160,
                             pBackBufferSurfaceDesc->Height-120 );
```

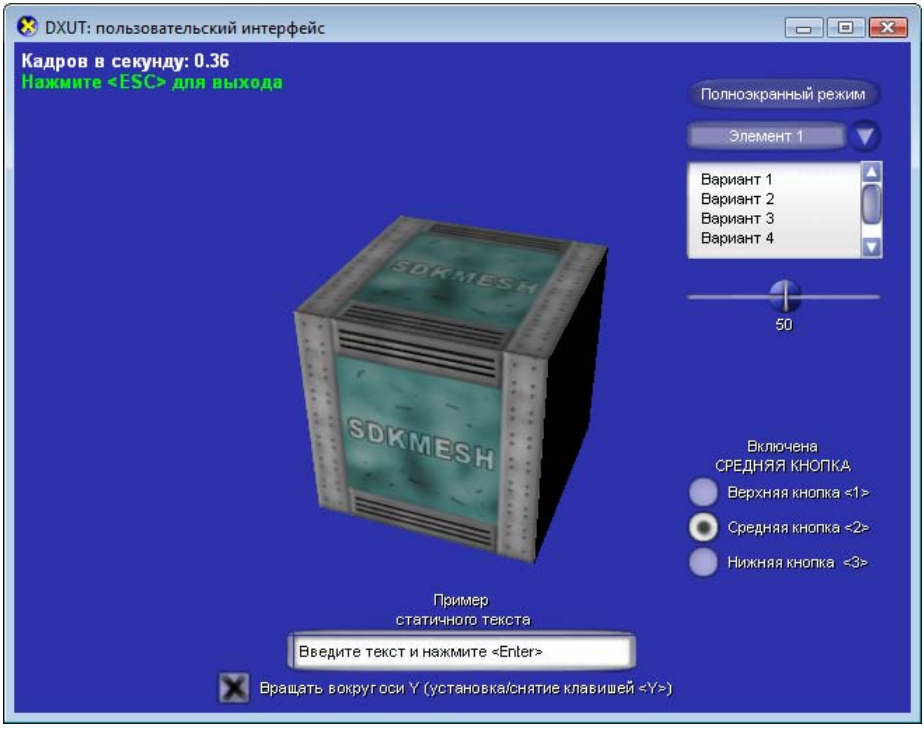

Рис. 11.21. Кнопки переключателей

Посмотрим, что у нас получилось. Компилируем и запускаем проект, видим появившиеся кнопки переключателя (рис. 11.21).

Щелкаем мышью на различных кнопках и обращаем внимание на то, как меняется сообщение элемента статичного текста. Испытываем действие "горячих клавиш", наблюдаем переключение выделения с одной кнопки на другую.

Проект с кнопками переключателей можно найти на компакт-диске в директории Glava11\DXUT_Radio.

На этом мы завершаем главу об элементах интерфейса пользователя в DXUT. Правда, с ними мы пока не прощаемся, в *гл. 12* они нам еще пригодятся.

Глава 12

Диалоговые окна

Мы с вами уже научились использовать все элементы интерфейса, однако пока не касались использования диалоговых окон. Мы их уже применяли, но только в качестве некоего "контейнера" для других элементов пользовательского интерфейса. А как быть, если нужно запросить у пользователя подтверждение какого-либо действия или вывести окно для настройки параметров? Рассмотрим работу с диалоговыми окнами на простом примере.

Создание диалогового окна

Пусть диалоговое окно в нашем примере будет содержать две кнопки: "ОК" и "Отмена", и мы будем его выводить для уточнения, действительно ли пользователь желает выйти из программы. Специально для этого окна мы добавим в программу кнопку "Выход".

Диалоговое окно будет вызываться только при нажатии кнопки "Выход", при нажатии клавиши <Esc> выход будет осуществлен немедленно. Существует возможность сделать так, чтобы диалоговое окно для подтверждения выводилось при любой попытке закрыть программу. Но для нас сейчас важен сам процесс создания и работы с диалоговыми окнами, так что отвлекаться на реализацию этой возможности мы не будем.

Само создание нового окна не представляет для нас ничего принципиально нового: точно так же, нужно объявить переменную типа `CDXUTDialog` и инициализировать элементы интерфейса окна при помощи тех же самых методов. Единственное отличие состоит в том, что мы должны задать размеры окна, цвет его фона, текст заголовка и положение на экране. Посмотрим, как все это делается. Сделаем копию последнего проекта, откроем ее и приступим к созданию нового диалогового окна.

Итак, объявляем глобальную переменную:

```
CDXUTDialog    g_ExitDialog;
```

Определим идентификаторы для кнопки "Выход" и элементов интерфейса нового диалогового окна: статичного текста и двух кнопок.

```
#define IDC_EXITBUTTON          13
#define IDC_EXIT_TEXT           14
#define IDC_EXIT_OK             15
#define IDC_EXIT_CANCEL         16
```

Следующим шагом идет инициализация, открываем функцию `InitApp()` и программируем ее. Как и у любого диалогового окна, в первую очередь мы вызываем метод `Init()`:

```
g_ExitDialog.Init( &g_DialogResourceManager );
```

Затем мы устанавливаем функцию для обработки событий. Оставим ту же самую функцию, что и для прочих элементов интерфейса:

```
g_ExitDialog.SetCallback ( OnGUIEvent );
```

Добавляем в основное окно программы кнопку "Выход":

```
g_UserInterface.AddButton( IDC_EXITBUTTON, L"Выход", 480, 400, 140, 25);
```

Теперь устанавливаем параметры нового диалогового окна. В первую очередь задаем его размеры:

```
g_ExitDialog.SetSize( 200, 120);
```

Как нетрудно догадаться, так мы задали окно размером 200×120 пикселов. Исходя из размеров, определим позицию, где оно будет выводиться. Давайте выведем его точно по центру главного окна программы:

```
g_ExitDialog.SetLocation(220, 180);
```

Включим отображение заголовка окна:

```
g_ExitDialog.EnableCaption ( TRUE );
```

Установим высоту заголовка:

```
g_ExitDialog.SetCaptionHeight( 15 );
```

Зададим выводимый в заголовке окна текст:

```
g_ExitDialog.SetCaptionText (L"Диалоговое окно");
```

Сделаем пока окно невидимым, сейчас оно нам на экране не нужно:

```
g_ExitDialog.SetVisible ( FALSE );
```

Задаем цвет фона диалогового окна:

```
g_ExitDialog.SetBackgroundColors ( D3DXCOLOR(1.0f, 0.2f, 0.2f, 1.0f) );
```

Переходим к элементам интерфейса, не забываем, что используется локальная система координат, связанная с новым диалоговым окном:

```
g_ExitDialog.AddStatic( IDC_EXIT_TEXT, L"Выйти из программы?",
                        30, 20, 150, 35 );
g_ExitDialog.AddButton ( IDC_EXIT_OK, L"OK", 20, 70, 60, 25, true );
g_ExitDialog.AddButton ( IDC_EXIT_CANCEL, L"Отмена", 120, 70, 60, 25 );
```

Все параметры мы задали, осталось запрограммировать вызов диалога на экран и действия кнопок самого диалога. Добавляем в функцию `OnGUIEvent()` вывод диалогового окна на экран по нажатию кнопки "Выход":

```
case IDC_EXITBUTTON:
    g_ExitDialog.SetVisible ( !g_ExitDialog.GetVisible() );
    break;
```

Если окно невидимо, при щелчке по кнопке мышью оно будет выведено на экран, а если оно уже видимо — то будет снова спрятано. Когда диалоговое окно стало видимым, появляется возможность использовать его элементы интерфейса. Так как мы спрашиваем согласие пользователя на выход из программы, при щелчке на кнопку "OK" программа должна закрыться, а если пользователь выбрал кнопку "Отмена", то мы опять скрываем диалоговое окно подтверждения и работа с программой продолжается. Вот как это выглядит на практике:

```
    // Если нажата кнопка OK - закрываем программу
  case IDC_EXIT_OK:
    DXUTShutdown();
    break;

  case IDC_EXIT_CANCEL:
    g_ExitDialog.SetVisible ( FALSE );
    break;
```

Для закрытия программы мы использовали функцию `DXUTShutdown()`.

Обеспечим прорисовку окна на экране, добавим в функцию `OnD3D10FrameRender()` вызов метода `OnRender()` после строки, в которой производится рисование интерфейса:

```
g_ExitDialog.OnRender( fElapsedTime );
```

Чтобы обеспечить нормальную работу интерфейса диалогового окна, нужно дать ему возможность обрабатывать сообщения системы. Добавим вызов метода `MsgProc()` диалогового окна перед обработкой сообщений камерой. Приведем новый вариант функции `MsgProc()` целиком:

```
LRESULT CALLBACK MsgProc( HWND hWnd, UINT uMsg, WPARAM wParam,
        LPARAM lParam, bool* pbNoFurtherProcessing, void* pUserContext )
```

```
{
  *pbNoFurtherProcessing =
           g_DialogResourceManager.MsgProc( hWnd, uMsg, wParam, lParam );
  if( *pbNoFurtherProcessing )
    return 0;

  // Обработка сообщений элементами интерфейса
  *pbNoFurtherProcessing =
           g_UserInterface.MsgProc( hWnd, uMsg, wParam, lParam );
  if( *pbNoFurtherProcessing )
    return 0;

  *pbNoFurtherProcessing =
           g_ExitDialog.MsgProc( hWnd, uMsg, wParam, lParam );
  if( *pbNoFurtherProcessing )
    return 0;

  // Передаем поступившие сообщения на обработку камере
  g_Camera.HandleMessages( hWnd, uMsg, wParam, lParam );
  return 0;
}
```

Основная работа с настройкой диалогового окна практически закончена. Осталось определить, как будет изменяться положение диалогового окна при изменении размеров основного окна программы. Оставим окно расположенным по центру, размеры окна по ширине и высоте, необходимые для вычисления координат его нового положения, получим при помощи методов `GetWidth()` и `GetHeight()` соответственно. Добавим в функцию `OnD3D10ResizedSwapChain()` строку, которая проводит это вычисление:

```
g_ExitDialog.SetLocation(
          (pBackBufferSurfaceDesc->Width-g_ExitDialog.GetWidth())/2,
          (pBackBufferSurfaceDesc->Height-g_ExitDialog.GetHeight())/2 );
```

Ну вот, мы готовы к тому, чтобы посмотреть на то, как работает наше диалоговое окно. Компилируем и запускаем проект. Когда программа запустится, мы заметим только добавление новой кнопки "Выход". Щелкнем по ней мышью. Появится диалоговое окно (рис. 12.1), предлагающее подтвердить выход из программы.

Щелкнем по кнопке "ОК", и программа закроется. Все работает, как мы и запланировали, все вроде бы нас устраивает.

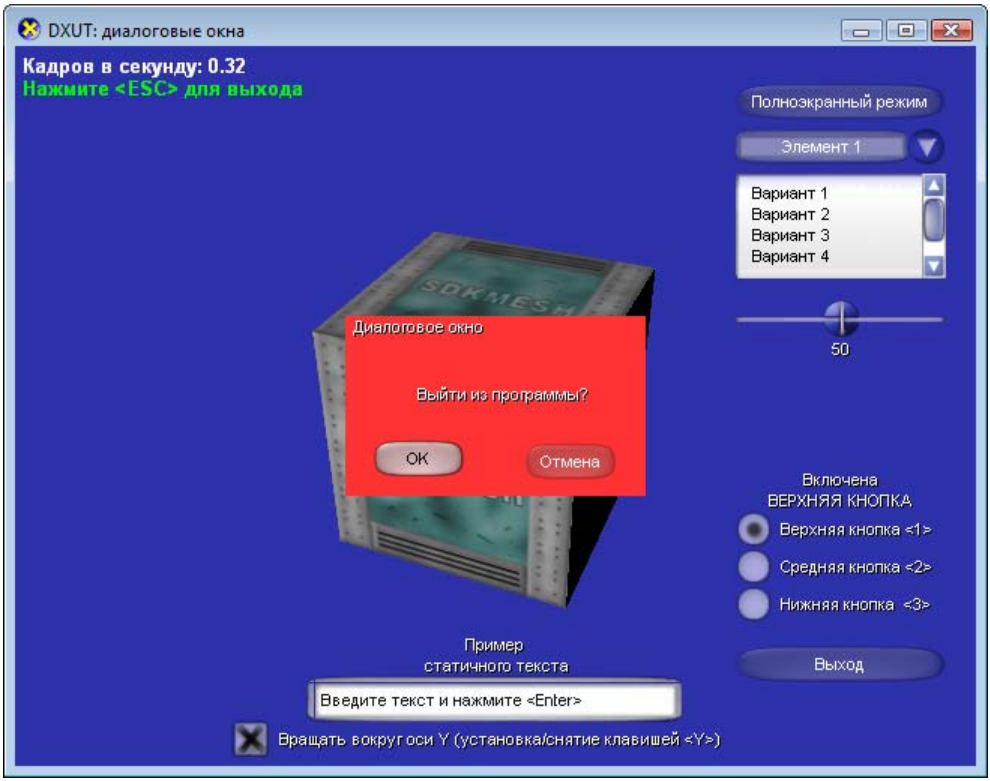

Рис. 12.1. Диалоговое окно подтверждения выхода из программы

Только вот почему и в заголовке диалогового окна, и в элементе статичного текста используется одинаковый шрифт? Текст вопроса, который задает диалоговое окно, как-то теряется, и вообще все окно от такого сочетания визуально только проигрывает. Нельзя ли как-то на это повлиять? Конечно же, можно. Добавим к имеющемуся шрифту нового диалогового окна еще один. Для этого вставим после вызова метода `Init()` вот такую строку:

```
g_ExitDialog.SetFont(1, L"Arial", 16, FW_BOLD );
```

Что все это означает? А означает это, что мы просим добавить шрифт, к которому можно потом обращаться по индексу 1, семейство символов "Arial", высота 16 пикселов, способ начертания — жирный. Применим этот шрифт к статичному тексту, содержащему вопрос: "Выйти из программы?". После инициализации статичного текста добавим строку:

```
g_ExitDialog.GetStatic( IDC_EXIT_TEXT )->GetElement( 0 )->iFont=1;
```

При помощи метода `GetElement()` мы получаем указатель на структуру данных, где хранятся свойства статичного текста, и меняем индекс шрифта

на тот, который недавно добавили. Запустим проект еще раз и щелкнем по кнопке "Выход". Диалоговое окно с новым шрифтом выглядит гораздо приятнее (рис. 12.2).

Готовый проект, в котором реализован вывод диалогового окна, находится на компакт-диске в директории Glava12\DXUT_Dialogue.

Создавать собственные диалоговые окна мы научились. Давайте теперь попробуем разобраться, как использовать в своей программе встроенное в DXUT диалоговое окно настройки параметров устройства Direct3D.

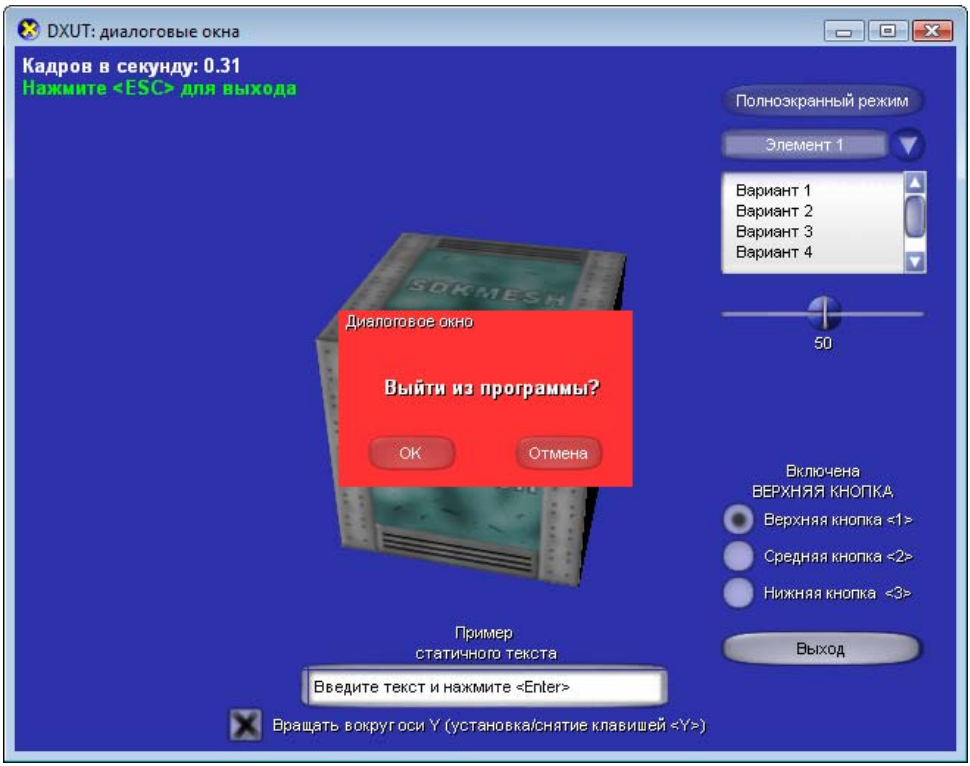

Рис. 12.2. Диалоговое окно с измененным шрифтом

Окно настройки параметров устройства Direct3D

С помощью этого окна мы можем изменить формат вторичного буфера, поменять параметры мультисэмплинга, переключить программу в полноэкранный режим либо вернуть ее в оконный. Использование этого окна в програм-

ме немного отличается от рассмотренного ранее. Давайте добавим это окно в наш проект, и будем вызывать его по щелчку на кнопке "Настройки", которую создадим специально для этого случая. Чтобы применять окно настройки параметров в своей программе, сначала нужно подключить заголовочный файл DXUTsettingsDlg.h. К тому же этот файл и файл DXUTsettingsDlg.cpp нужно добавить в проект.

Объявляем глобальную переменную для диалогового окна настроек:

```
CD3DSettingsDlg    g_D3DSettingsDlg;
```

Определяем идентификатор для кнопки "Настройки":

```
#define IDC_DEVSETTINGS    17
```

Инициализация диалогового окна настроек не отличается от инициализации обычного диалогового окна. Добавляем в функцию `InitApp()` строку с вызовом метода `Init()`:

```
g_D3DSettingsDlg.Init( &g_DialogResourceManager );
```

На этом инициализация закончена: все необходимые элементы интерфейса окна настроек добавляются автоматически, нам осталось только ввести кнопку, с помощью которой будет активироваться диалоговое окно:

```
g_UserInterface.AddButton( IDC_DEVSETTINGS, L"Настройки",
                           480, 230, 140, 25 );
```

Поскольку диалоговое окно связано с устройством Direct3D, необходимо в функцию `OnD3D10CreateDevice()` добавить вызов метода `OnD3D10CreateDevice()` окна настроек. Вставляем его после вызова аналогичного метода менеджера ресурсов:

```
V_RETURN( g_D3DSettingsDlg.OnD3D10CreateDevice( pd3dDevice ) );
```

Обеспечиваем активирование окна при помощи созданной кнопки "Настройки". Для этого добавляем в функцию `OnGUIEvent()` соответствующую ветвь оператора `switch`:

```
case IDC_DEVSETTINGS:
  g_D3DSettingsDlg.SetActive( !g_D3DSettingsDlg.IsActive() );
  break;
```

Метод `IsActive()` используется для того, чтобы узнать, активировано ли окно в данный момент. Так как окно настроек всегда полностью занимает всю доступную площадь основного окна, то, когда оно активировано, другие объекты можно на экран не выводить. Поэтому в функцию `OnD3D10FrameRender()` сразу после очистки буфера визуализации и шаблонного буфера глубины добавляем такой текст:

```
if( g_D3DSettingsDlg.IsActive() )
  {
```

```
    g_D3DSettingsDlg.OnRender( fElapsedTime );
    return;
  }
```

В случае если окно активировано, будет прорисовано только окно настроек, а остальная часть функции визуализации будет проигнорирована. Похожим образом нужно поступить и в функции обработки сообщений, вот как должна выглядеть функция `MsgProc` в этом случае:

```
LRESULT CALLBACK MsgProc( HWND hWnd, UINT uMsg, WPARAM wParam,
LPARAM lParam, bool* pbNoFurtherProcessing, void* pUserContext )
{
  *pbNoFurtherProcessing =
          g_DialogResourceManager.MsgProc( hWnd, uMsg, wParam, lParam );
  if( *pbNoFurtherProcessing )
    return 0;

  // Если открыто окно настроек, передать сообщения на обработку
  if( g_D3DSettingsDlg.IsActive() )
  {
    g_D3DSettingsDlg.MsgProc( hWnd, uMsg, wParam, lParam );
    return 0;
  }

  // Обработка сообщений элементами интерфейса
  *pbNoFurtherProcessing =
          g_UserInterface.MsgProc( hWnd, uMsg, wParam, lParam );
  if( *pbNoFurtherProcessing )
    return 0;

  *pbNoFurtherProcessing =
          g_ExitDialog.MsgProc( hWnd, uMsg, wParam, lParam );
  if( *pbNoFurtherProcessing )
    return 0;

  // Передаем поступившие сообщения на обработку камере
  g_Camera.HandleMessages( hWnd, uMsg, wParam, lParam );
  return 0;
}
```

Для обеспечения реакции окна настроек на изменение размеров основного окна программы нужно из функции `OnD3D10ResizedSwapChain()` вызывать метод `OnD3D10ResizedSwapChain()`. Делается это так:

```
V_RETURN( g_D3DSettingsDlg.OnD3D10ResizedSwapChain( pd3dDevice,
                                                    pBackBufferSurfaceDesc ) );
```

Обеспечим сохранение положения кнопки "Настройки" относительно других элементов интерфейса, будем располагать ее строго по середине высоты окна у его правой границы:

```
g_UserInterface.GetControl( IDC_DEVSETTINGS )->SetLocation(
          pBackBufferSurfaceDesc->Width-160,
          (pBackBufferSurfaceDesc->Height-
             g_UserInterface.GetControl(IDC_DEVSETTINGS )->m_height)/2 );
```

При выходе из программы нужно корректно освободить все занятые ресурсы, в этом нам поможет метод `OnD3D10DestroyDevice()`. Вот как должна выглядеть функция, в которой он вызывается:

```
void CALLBACK OnD3D10DestroyDevice( void* pUserContext )
{
  g_DialogResourceManager.OnD3D10DestroyDevice();
  g_D3DSettingsDlg.OnD3D10DestroyDevice();
  DXUTGetGlobalResourceCache().OnDestroyDevice();
  SAFE_RELEASE( g_pFont );
  SAFE_RELEASE( g_pSprite );
  SAFE_DELETE( g_pTxtHelper );
  SAFE_RELEASE( g_pVertexLayout );
  SAFE_RELEASE( g_pEffect );
  g_Mesh.Destroy();
}
```

Запустим проект и проверим окно настроек в работе. В основном окне программы добавилась новая кнопка для вызова окна настроек (рис. 12.3).

Щелкнем на этой кнопке мышью и получим окно для настройки параметров устройства Direct3D (рис. 12.4). Для проверки его работоспособности пробуем с его помощью переключить программу в полноэкранный режим. Программа должна переключиться в полноэкранный режим.

Готовый проект, использующий окно настроек, можно найти на компакт-диске в директории Glava12\DXUT_DevSettings.

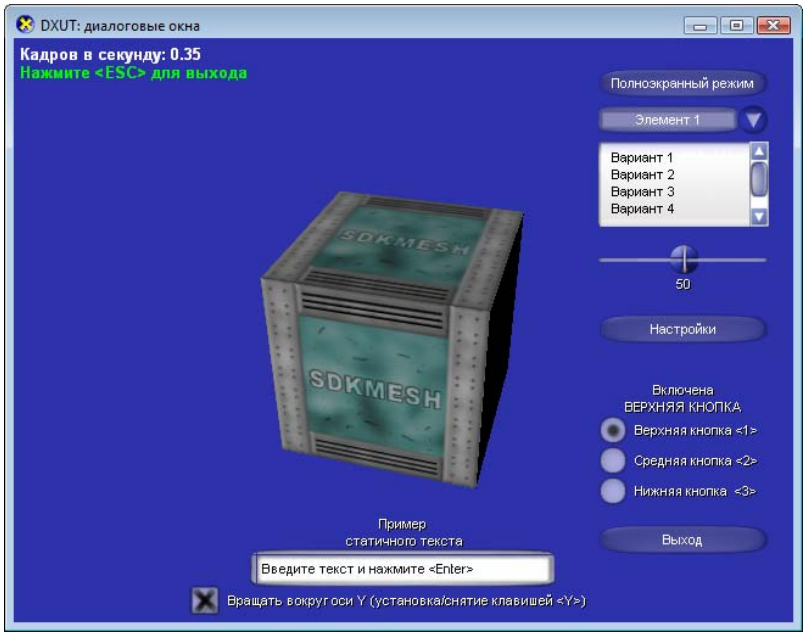

Рис. 12.3. Кнопка для вызова окна настроек

Рис. 12.4. Окно настройки параметров устройства Direct3D

ПРИЛОЖЕНИЯ

Приложение 1

Краткий справочник по DirectX10

В этом приложении находится справочная информация по функциям DirectX 10. Этот справочник, конечно, не сможет заменить документацию, поставляемую вместе с DirectX SDK, но он поможет лучше разобраться с примерами программ, которые встречаются в книге. Справочник содержит три основных раздела:

- интерфейсы;
- функции;
- структуры.

Интерфейсы

DirectX 10 определяет большое количество интерфейсов. Мы опишем лишь наиболее часто используемые из них.

ID3D10Blob

Используется для работы с данными произвольной длины. Создаваемые буферы можно использовать для хранения данных о вершинах, смежных объектах, о материалах при оптимизации моделей и считывании данных. Кроме того, эти объекты используются для получения объектного кода и сообщений об ошибках при компиляции вершинных, геометрических и пиксельных шейдеров. Интерфейс содержит следующие методы.

ID3D10Blob::GetBufferPointer

Возвращает указатель на буфер данных. Прототип метода:

```
LPVOID GetBufferPointer();
```

Параметры: отсутствуют.

ID3D10Blob::GetBufferSize

Возвращает размер буфера данных. Прототип метода:

```
SIZE_T GetBufferSize();
```

Параметры: отсутствуют.

ID3D10Buffer

Интерфейс управляет буферным ресурсом, который представляет собой неструктурированный фрагмент памяти. Такие буферы, как правило, хранят данные о вершинах либо об индексах вершин. Возможно создание трех типов буферов: вершинный буфер, индексный буфер и буфер констант шейдера. Для создания буферного ресурса необходимо вызвать метод `ID3D10Device::CreateBuffer()`. Чтобы получить доступ к данным буфера, сначала нужно связать его с графическим конвейером. Буфер можно связать со стадией компоновки входных данных (input-assembler stage) при помощи методов `ID3D10Device::IASetVertexBuffers()` и `ID3D10Device::IASetIndexBuffer()`. Со стадией вывода данных буфер можно связать при помощи метода `ID3D10Device::SOSetTargets()`. Интерфейс содержит следующие методы.

ID3D10Buffer::GetDesc

Получает свойства буферного ресурса. Прототип метода:

```
void GetDesc(
  D3D10_BUFFER_DESC *pDesc
);
```

Параметр:

- `pDesc` — указатель на структуру типа `D3D10_BUFFER_DESC`, в которую метод поместит свойства буферного ресурса. Значение `NULL` недопустимо.

ID3D10Buffer::Map

Получает указатель на данные, хранящиеся в буферном ресурсе, и блокирует доступ к этим данным процессора видеокарты. Прототип метода:

```
HRESULT Map(
  D3D10_MAP MapType,
  UINT MapFlags,
  void **ppData
);
```

Параметры:

- `MapType` — флаг, определяющий разрешения на доступ процессору видеокарты на чтение из ресурса и запись в ресурс (см. `D3D10_MAP`);

- `MapFlags` — флаг, определяющий, как следует поступить центральному процессору, если процессор видеокарты занят (см. `D3D_MAP_FLAG`), этот флаг необязательный;
- `ppData` — указатель на данные буферного ресурса.

Возвращаемые значения:

Если в процессе выполнения не было ошибок, метод возвращает значение `S_OK`. По каким причинам могут возникнуть ошибки:

- Если параметр `MapType` содержит флаг `D3D10_MAP_FLAG_DO_NOT_WAIT` и процессор видеокарты еще работает с ресурсом, метод вернет код ошибки `E_WASSTILLRENDERING`.
- Метод возвращает код ошибки `E_DEVICEREMOVED`, если `MapType` включает любой флаг, разрешающий чтение и при этом аппаратное устройство (видеокарта) было удалено из системы.

ID3D10Buffer::Unmap

Освобождает указатель на ресурс, полученный с помощью метода `ID3D10Buffer::Map()` и вновь разрешает процессору видеокарты доступ к буферу. Прототип метода:

```
void Unmap();
```

Параметры: отсутствуют.

ID3D10DepthStencilView

Интерфейс представления данных как шаблонного буфера глубины управляет текстурным ресурсом, который используется во время проверки расстояния от пиксела до камеры и необходимости его обновления (depth-stencil test). Для создания интерфейса используется метод `ID3D10Device::CreateDepthStencilView()`. Интерфейс содержит следующий метод.

ID3D10DepthStencilView::GetDesc

Получает параметры представления данных как шаблонного буфера глубины. Прототип метода:

```
void GetDesc(
  D3D10_DEPTH_STENCIL_VIEW_DESC *pDesc
);
```

Параметр:

- `pDesc` — указатель на структуру, в которую метод поместит свойства представления данных как шаблонного буфера глубины.

ID3D10Device

Устройство представляет собой виртуальный адаптер Direct3D 10, его используют для прорисовки кадров и для создания ресурсов Direct3D 10. Некоторые методы интерфейса описаны ниже.

ID3D10Device::ClearDepthStencilView

Очищает ресурс, принадлежащий шаблонному буферу глубины. Прототип метода:

```
void ClearDepthStencilView(
  ID3D10DepthStencilView *pDepthStencilView,
  UINT ClearFlags,
  FLOAT Depth,
  UINT8 Stencil
);
```

Параметры:

- `pDepthStencilView` — указатель на шаблонный буфер глубины, который нужно очистить;
- `ClearFlags` — флаг, указывающий, какую часть буфера очищать;
- `Depth` — значение, которым заполняется буфер глубины, значение будет приведено к интервалу от 0 до 1;
- `Stencil` — значение, которым будет заполнен шаблонный буфер.

ID3D10Device::ClearRenderTargetView

Очищает буфер визуализации, устанавливая все пикселы в один указанный цвет. Прототип метода:

```
void ClearRenderTargetView(
  ID3D10RenderTargetView *pRenderTargetView,
  const FLOAT ColorRGBA[4]
);
```

Параметры:

- `ID3D10RenderTargetView` — указатель на буфер визуализации;
- `ColorRGBA` — массив из четырех элементов, представляющих компоненты цвета для заполнения буфера визуализации.

ID3D10Device::ClearState

Восстанавливает настройки устройства по умолчанию, возвращая устройство в состояние, в котором оно прибывало сразу после создания. Прототип метода:

```
void ClearState();
```

Параметры: отсутствуют.

ID3D10Device::CreateBuffer

Создает буфер (вершинный буфер, индексный буфер либо буфер констант шейдера) Прототип метода:

```
HRESULT CreateBuffer(
  const D3D10_BUFFER_DESC *pDesc,
  const D3D10_SUBRESOURCE_DATA *pInitialData,
  ID3D10Buffer **ppBuffer
);
```

Параметры:

- `pDesc` — указатель на структуру типа `D3D10_BUFFER_DESC`, содержащую описание свойств буфера;
- `pInitialData` — указатель на данные для инициализации буфера (см. `D3D10_SUBRESOURCE_DATA`). Если требуется только выделить память, можно использовать значение `NULL`;
- `ppBuffer` — адрес указателя, который будет установлен на созданный буфер.

ID3D10Device::CreateDepthStencilView

Создает представление данных как шаблонного буфера глубины для доступа к ним. Прототип метода:

```
HRESULT CreateDepthStencilView(
  ID3D10Resource *pResource,
  const D3D10_DEPTH_STENCIL_VIEW_DESC *pDesc,
  ID3D10DepthStencilView **ppDepthStencilView
);
```

Параметры:

- `pResource` — указатель на ресурс, который будет использоваться в качестве шаблонного буфера глубины. Этот ресурс должен быть создан с использованием флага `D3D10_BIND_DEPTH_STENCIL`.
- `pDesc` — указатель на описание свойств представления данных как шаблонного буфера глубины (см. `D3D10_DEPTH_STENCIL_VIEW_DESC`). Можно

установить этот параметр в NULL, если требуется создать представление данных для доступа к нулевому уровню мипмаппинга и ко всему объему ресурса. При этом будет использоваться формат ресурса, указанный при его создании.

- ppDepthStencilView — адрес указателя на интерфейс ID3D10DepthStencilView.

ID3D10Device::CreateInputLayout

Создает объект входных данных для описания входных данных, передаваемых на этап компоновки входных данных конвейера. Прототип метода:

```
HRESULT CreateInputLayout(
  const D3D10_INPUT_ELEMENT_DESC *pInputElementDescs,
  UINT NumElements,
  const void *pShaderBytecodeWithInputSignature,
  SIZE_T BytecodeLength,
  ID3D10InputLayout **ppInputLayout
);
```

Параметры:

- pInputElementDescs — массив, содержащий описания (при помощи структуры D3D10_INPUT_ELEMENT_DESC) типов данных, передаваемых на первую стадию графического конвейера;
- NumElements — количество описаний типов данных, то есть количество элементов в массиве;
- pShaderBytecodeWithInputSignature — указатель на скомпилированный шейдер, содержащий сигнатуру входных данных шейдера;
- BytecodeLength — размер скомпилированного шейдера в байтах;
- ppInputLayout — адрес указателя, в который будет записана ссылка на создаваемый объект входных данных.

ID3D10Device::CreateRenderTargetView

Создает представление данных как буфера визуализации для доступа к данным ресурса. Прототип метода:

```
HRESULT CreateRenderTargetView(
  ID3D10Resource *pResource,
  const D3D10_RENDER_TARGET_VIEW_DESC *pDesc,
  ID3D10RenderTargetView **ppRTView
);
```

Параметры:

- `pResource` — указатель на содержащий данные ресурс;
- `pDesc` — указатель на структуру типа `D3D10_RENDER_TARGET_VIEW_DESC`, содержащую описание параметров буфера визуализации (значение `NULL` означает использование параметров по умолчанию);
- `ppRTView` — адрес указателя, который будет установлен методом на созданное представление данных.

ID3D10Device::Draw

Рисует примитивы без использования индексов и создания экземпляров (instancing). Прототип метода:

```
void Draw(
  UINT VertexCount,
  UINT StartVertexLocation
);
```

Параметры:

- `VertexCount` — количество вершин, которое необходимо нарисовать;
- `StartVertexLocation` — индекс вершины, с которой начинается рисование.

ID3D10Device::DrawIndexed

Рисует примитивы с использованием индексов, без создания экземпляров (instancing). Прототип метода:

```
void DrawIndexed(
  UINT IndexCount,
  UINT StartIndexLocation,
  INT BaseVertexLocation
);
```

Параметры:

- `IndexCount` — количество индексов вершин для прорисовки;
- `StartIndexLocation` — индекс первого индекса вершины, с которого начинается прорисовка (полезно, если в одном индексном буфере хранятся описания нескольких объектов);
- `BaseVertexLocation` — индекс первой вершины в вершинном буфере, который добавляется к каждому индексу вершины, чтобы получить соответствующий элемент из вершинного буфера, может принимать отрицатель-

ные значения (он также используется при хранении в вершинном буфере нескольких объектов).

ID3D10Device::OMSetRenderTargets

Связывает один или несколько буферов визуализации и шаблонный буфер глубины со стадией компоновки входных данных графического конвейера. Прототип метода:

```
void OMSetRenderTargets(
  UINT NumViews,
  ID3D10RenderTargetView *const *ppRenderTargetViews,
  ID3D10DepthStencilView *pDepthStencilView
);
```

Параметры:

- `NumViews` — количество представлений данных для связывания с устройством;
- `ppRenderTargetViews` — указатель на массив представлений данных для буферов визуализации, которые нужно связать с устройством;
- `pDepthStencilView` — указатель на представление данных как шаблонного буфера глубины.

ID3D10Device::IASetIndexBuffer

Связывает индексный буфер со стадией компоновки входных данных графического конвейера. Прототип метода:

```
void IASetIndexBuffer(
  ID3D10Buffer *pIndexBuffer,
  DXGI_FORMAT Format,
  UINT Offset
);
```

Параметры:

- `pIndexBuffer` — указатель на буфер, содержащий индексы вершин;
- `Format` — флаг, определяющий формат данных индексного буфера, допустимыми являются только два значения: `DXGI_FORMAT_R16_UINT` и `DXGI_FORMAT_R32_UINT`;
- `Offset` — смещение в байтах от начала индексного буфера до первой используемой вершины.

ID3D10Device::IASetInputLayout

Связывает объект входных данных со стадией компоновки входных данных графического конвейера. Прототип метода:

```
void IASetInputLayout(
  ID3D10InputLayout *pInputLayout
);
```

Параметр:

- `pInputLayout` — указатель на объект входных данных (см. `ID3D10InputLayout`), который описывает буферы входных данных, считываемые на этапе компоновки входных данных графического конвейера.

ID3D10Device::IASetPrimitiveTopology

Связывает информацию о типе примитивов с последовательностью данных, которая описывает входные данные для первой стадии графического конвейера. Прототип метода:

```
void IASetPrimitiveTopology(
  D3D10_PRIMITIVE_TOPOLOGY Topology
);
```

Параметр:

- `Topology` — тип примитива и способ вывода примитивов.

ID3D10Effect

Интерфейс управляет набором режимов, ресурсов и шейдеров для применения эффекта визуализации. Некоторые методы интерфейса приведены ниже.

ID3D10Effect::GetTechniqueByName

Извлекает технику отображения по ее имени. Прототип метода:

```
ID3D10EffectTechnique* GetTechniqueByName(
  LPCSTR Name
);
```

Параметр:

- `Name` — имя техники отображения.

ID3D10Effect::GetVariableByName

Получает доступ к переменной шейдера по ее имени. Прототип метода:

```
ID3D10EffectVariable* GetVariableByName(
  LPCSTR Name
);
```

Параметр:

- `Name` — имя переменной.

ID3D10EffectMatrixVariable

Интерфейс для доступа к матричным переменным шейдеров. Некоторые методы интерфейса представлены ниже.

ID3D10EffectMatrixVariable::GetMatrix

Извлекает значения из матричной переменной шейдера. Прототип метода:

```
HRESULT GetMatrix(
  float *pData
);
```

Параметр:

- `pData` — указатель на первый элемент матрицы, куда извлекаются значения.

ID3D10EffectMatrixVariable::SetMatrix

Устанавливает значения матричной переменной шейдера. Прототип метода:

```
HRESULT SetMatrix(
  float *pData
);
```

Параметр:

- `pData` — указатель на первый элемент матрицы, откуда передаются значения в переменную шейдера.

ID3D10EffectScalarVariable

Интерфейс для доступа к скалярным переменным шейдеров. Некоторые методы интерфейса представлены ниже.

ID3D10EffectScalarVariable::SetBool

Устанавливает значение булевой переменной шейдера. Прототип метода:

```
HRESULT SetBool(
  BOOL Value
);
```

Параметр:

- `Value` — значение, которое нужно установить.

ID3D10EffectScalarVariable::SetFloat

Установить значение переменной шейдера с плавающей точкой. Прототип метода:

```
HRESULT SetFloat(
  float Value
);
```

Параметр:

- `Value` — значение, которое нужно установить.

ID3D10EffectScalarVariable::SetFloatArray

Устанавливает значения массива из элементов с плавающей точкой. Прототип метода:

```
HRESULT SetFloatArray(
  float *pData,
  UINT Offset,
  UINT Count
);
```

Параметры:

- `pData` — указатель на первое значение массива;
- `Offset` — смещение (представленное в элементах массива) от первого элемента массива до элемента, с которого нужно начинать установку значений;
- `Count` — количество элементов в массиве.

ID3D10EffectScalarVariable::SetInt

Устанавливает значение целочисленной переменной шейдера. Прототип метода:

```
HRESULT SetInt(
  int Value
);
```

Параметр:

- `Value` — значение, которое нужно установить.

ID3D10EffectScalarVariable::SetIntArray

Устанавливает значения массива из целочисленных элементов. Прототип метода:

```
HRESULT SetIntArray(
  int *pData,
```

```
  UINT Offset,
  UINT Count
);
```

Параметры:

- `pData` — указатель на первое значение массива;
- `Offset` — смещение (представленное в элементах массива) от первого элемента массива до элемента, с которого нужно начинать установку значений;
- `Count` — количество элементов в массиве.

ID3D10EffectTechnique

Интерфейс представляет собой набор проходов визуализации. Некоторые из методов интерфейса представлены ниже.

ID3D10EffectTechnique::GetPassByIndex

Получает доступ к проходу визуализации по его индексу. Прототип метода:

```
ID3D10EffectPass* GetPassByIndex(
  UINT Index
);
```

Параметр:

- `Index` — индекс прохода (отсчитывается от нуля).

ID3D10EffectTechnique::GetPassByName

Получает доступ к проходу визуализации по его имени. Прототип метода:

```
ID3D10EffectPass* GetPassByName(
  LPCSTR Name
);
```

Параметр:

- `Name` — имя прохода.

ID3D10EffectVectorVariable

Интерфейс для доступа к векторным переменным шейдеров, содержащих четыре координаты. Некоторые из методов интерфейса приведены далее.

ID3D10EffectVectorVariable::SetFloatVector

Устанавливает значения четырехкомпонентного вектора, содержащего значения с плавающей точкой. Прототип метода:

```
HRESULT SetFloatVector(
  float *pData
);
```

Параметр:

- `pData` — указатель на первый компонент.

ID3D10EffectVectorVariable::SetFloatVectorArray

Устанавливает значения массива из четырехкомпонентных векторов, содержащих значения с плавающей точкой. Прототип метода:

```
HRESULT SetFloatVectorArray(
  float *pData,
  UINT Offset,
  UINT Count
);
```

Параметры:

- `pData` — указатель на первый компонент первого вектора в массиве.
- `Offset` — смещение (представленное в элементах массива) от начала массива до вектора, с которого нужно начинать установку значений.
- `Count` — количество элементов в массиве.

ID3D10InputLayout

Интерфейс считывает данные для этапа компоновки входных данных графического конвейера. Интерфейс не содержит дополнительных методов. Для создания интерфейса используется метод `ID3D10Device::CreateInputLayout()`, для того чтобы связать интерфейс с графическим конвейером, применяется метод `ID3D10Device::IASetInputLayout()`.

ID3D10RenderTargetView

Интерфейс определяет подресурсы буфера визуализации, к которым возможен доступ при выполнении прорисовки сцены. Интерфейс содержит следующий метод.

ID3D10RenderTargetView::GetDesc

Получить описание буфера визуализации. Прототип метода:

```
void GetDesc(
  D3D10_RENDER_TARGET_VIEW_DESC *pDesc
);
```

Параметр:

- pDesc — указатель на структуру, которая будет содержать свойства буфера визуализации (см. D3D10_RENDER_TARGET_VIEW_DESC).

ID3D10ShaderResourceView

Интерфейс определяет подресурсы, которые доступны шейдеру при прорисовке трехмерной сцены. В качестве примеров ресурсов шейдера можно назвать буфер констант, текстурный буфер, текстуру или сэмплер. Интерфейс содержит следующий метод.

ID3D10ShaderResourceView::GetDesc

Получает описание представления данных как ресурса шейдера. Прототип метода:

```
void GetDesc(
  D3D10_SHADER_RESOURCE_VIEW_DESC *pDesc
);
```

Параметр:

- pDesc — указатель на структуру, в которую будут помещены свойства представления данных как ресурса шейдера (см. D3D10_SHADER_RESOURCE_VIEW_DESC).

ID3DX10Font

Интерфейс включает в себя ресурсы, необходимые для вывода на экран текста указанным шрифтом и при помощи указанного устройства. Один из методов интерфейса приведен ниже.

ID3DX10Font::DrawText

Выводит отформатированный текст. Метод поддерживает как строки с символами ANSI, так и строки с символами Unicode. Прототип метода:

```
INT DrawText(
  ID3DX10Sprite *pSprite,
```

```
  LPCSTR pString,
  INT Count,
  LPRECT pRect,
  UINT Format,
  D3DXCOLOR Color
);
```

Параметры:

- `pSprite` — указатель на спрайтовый объект `ID3DX10Sprite`, который будет отображать строку символов, если он имеет значение `NULL`, Direct3D выведет строку символов, используя собственный спрайтовый объект;
- `pString` — указатель на строку символов, которую необходимо вывести;
- `Count` — количество символов в выводимой строке, значение –1 указывает, что `pString` указывает на строку с завершающим нулем и количество символов нужно подсчитать автоматически;
- `pRect` — указатель на прямоугольник, относительно которого осуществляется форматирование текста;
- `Format` — флаг, который определяет способ форматирования текста и может представлять собой любую комбинацию специальных флагов, которые приведены в табл. П1.1;
- `Color` — цвет текста.

Таблица П1.1. *Флаги для указания способа форматирования текста*

Значение	Описание
`DT_BOTTOM`	Выравнивает текст по нижней границе прямоугольника, нужно использовать в сочетании с флагом `DT_SINGLELINE`
`DT_CALCRECT`	Автоматически вычисляет ширину и высоту прямоугольника, исходя из длины указанной строки для вывода. Если выводится многострочный текст, `ID3DX10Font::DrawText` сохранит ширину прямоугольника, заданного параметром `pRect`, и изменит его высоту так, чтобы он охватывал последнюю строку текста. Если же текст представляет собой единственную строку, метод поменяет правую границу прямоугольника таким образом, чтобы последний символ строки находился внутри прямоугольника. В обоих случаях `ID3DX10Font::DrawText` возвращает высоту отформатированного текста, но вывод текста не осуществляет
`DT_CENTER`	Выравнивает текст в горизонтальном направлении по центру прямоугольника
`DT_EXPANDTABS`	Заменяет символы табуляции пробелами. По умолчанию один символ табуляции эквивалентен восьми пробелам

Таблица П1.1 *(окончание)*

Значение	Описание
DT_LEFT	Выравнивает текст по левому краю прямоугольника
DT_NOCLIP	Рисует текст без отсечения, при использовании этого флага метод работает несколько быстрее
DT_RIGHT	Выравнивает текст по правому краю прямоугольника
DT_RTLREADING	Выводит текст в режиме "справа налево", если выбран соответствующий шрифт (арабский или еврейский), используется для вывода двунаправленных текстов. По умолчанию направление вывода всего остального текста — "слева направо"
DT_SINGLELINE	Отображает весь текст вытянутым в одну строку. Имеющиеся в тексте символы возврата каретки и перевода строки не разрывают строку
DT_TOP	Выравнивает текст по верхней границе прямоугольника
DT_VCENTER	Выравнивает текст по центру прямоугольника в вертикальном направлении (только для текста в одну строку)
DT_WORDBREAK	Выполняет перенос по словам. Очередное слово в тексте автоматически переносится на следующую строку, если в текущей строке оно может выйти за границу прямоугольника, заданного параметром pRect. Имеющиеся в тексте символы возврата каретки и перевода строки также разрывают строку

ID3DX10Sprite

Интерфейс предоставляет набор методов, облегчающих вывод спрайтов с использованием Direct3D, имеется возможность работать сразу с большим количеством спрайтов. Интерфейс содержит следующие методы.

ID3DX10Sprite::Begin

Подготавливает устройство Direct3D 10 к рисованию спрайтов. Прототип метода:

```
HRESULT Begin(
  UINT flags
);
```

Параметр:

- flags — флаги, управляющие способом прорисовки спрайтов (см. D3DX10_SPRITE_FLAG).

ID3DX10Sprite::DrawSpritesBuffered

Добавляет массив спрайтов к очереди спрайтов на прорисовку. Метод необходимо вызывать между вызовами методов ID3DX10Sprite::Begin()

и `ID3DX10Sprite::End()`, кроме того, перед вызовом `ID3DX10Sprite::End()` нужно вызвать метод `ID3DX10Sprite::Flush()`, чтобы отправить всю очередь спрайтов на прорисовку. Прототип метода:

```
HRESULT DrawSpritesBuffered(
  D3DX10_SPRITE *pSprites,
  UINT cSprites
);
```

Параметры:

- `pSprites` — массив спрайтов для вывода (см. `D3DX10_SPRITE`);
- `cSprites` — количество спрайтов в массиве `pSprites`.

ID3DX10Sprite::DrawSpritesImmediate

Производит прорисовку массива спрайтов, но в отличие от метода `ID3DX10Sprite::DrawSpritesBuffered()` спрайты сразу же отправляются в устройство Direct3D. Метод также необходимо вызывать между вызовами методов `ID3DX10Sprite::Begin()` и `ID3DX10Sprite::End()`. Прототип метода:

```
HRESULT DrawSpritesImmediate(
  D3DX10_SPRITE *pSprites,
  UINT cSprites,
  UINT cbSprite,
  UINT flags
);
```

Параметры:

- `pSprites` — массив спрайтов для вывода (см. `D3DX10_SPRITE`);
- `cSprites` — количество спрайтов в `pSprites`;
- `cbSprite` — размер спрайтовой структуры, которая передается в `pSprites`, нулевое значение эквивалентно передаче `sizeof(D3DX10_SPRITE)`;
- `flags` — зарезервировано.

ID3DX10Sprite::End

Вызывается после метода `ID3DX10Sprite::Flush()`, возвращает состояние устройства, которое оно имело до вызова метода `ID3DX10Sprite::Begin()`. Прототип метода:

```
HRESULT End();
```

Параметры: отсутствуют.

ID3DX10Sprite::Flush

Отправляет всю очередь спрайтов на прорисовку устройству Direct3D. Прототип метода:

```
HRESULT Flush();
```

Параметры: отсутствуют.

ID3DX10Sprite::GetDevice

Извлекает указатель на устройство Direct3D, связанное со спрайтовым объектом. Прототип метода:

```
HRESULT GetDevice(
  ID3D10Device **ppDevice
);
```

Параметр:

- `ppDevice` — адрес указателя на интерфейс `ID3D10Device`, представляющий устройство Direct3D, связанное со спрайтовым объектом.

ID3DX10Sprite::GetProjectionTransform

Извлекает матрицу проекции, которая применяется ко всем спрайтам. Прототип метода:

```
HRESULT GetProjectionTransform(
  D3DXMATRIX *pProjectionTransform
);
```

Параметр:

- `pProjectionTransform` — указатель на структуру, куда нужно поместить матрицу проекции спрайтов.

ID3DX10Sprite::GetViewTransform

Извлечь видовое преобразование, применяемое ко всем спрайтам. Прототип метода:

```
HRESULT GetViewTransform(
  D3DXMATRIX *pViewTransform
);
```

Параметр:

- `pViewTransform` — указатель на структуру, куда нужно поместить видовую матрицу для спрайтов.

ID3DX10Sprite::SetProjectionTransform

Задает матрицу проекции для всех спрайтов. Прототип метода:

```
HRESULT SetProjectionTransform(
  D3DXMATRIX *pProjectionTransform
);
```

Параметр:

- `pProjectionTransform` — указатель на матрицу, которую необходимо использовать для всех спрайтов.

ID3DX10Sprite::SetViewTransform

Задает видовое преобразование координат, которое применяется ко всем спрайтам. Прототип метода:

```
HRESULT SetViewTransform(
  D3DXMATRIX *pViewTransform
);
```

Параметр:

- `pViewTransform` — указатель на матрицу, которую необходимо использовать для всех спрайтов.

IDXGISwapChain

Интерфейс цепочки переключений предоставляет один или более буферов, в которых хранятся прорисованные объекты до вывода на экран. Ниже описаны некоторые его методы.

IDXGISwapChain::GetBuffer

Получает доступ к одному из вторичных буферов цепочки переключений. Прототип метода:

```
HRESULT GetBuffer(
  UINT Buffer,
  REFIID riid,
  void **ppSurface
);
```

Параметры:

- `Buffer` — индекс вторичного буфера, к которому нужно получить доступ;
- `riid` — идентификатор интерфейса (GUID), который используется для выполнения действий с буфером;

- `ppSurface` — адрес указателя на интерфейс вторичного буфера, метод установит этот указатель на вторичный буфер.

IDXGISwapChain::Present

Отображает прорисованную графическую сцену на экране. Прототип метода:

```
HRESULT Present(
  UINT SyncInterval,
  UINT Flags
);
```

Параметры:

- `SyncInterval` — интервал синхронизации изображения, возможные значения:
  - `0` — без синхронизации, вывод происходит немедленно;
  - `1..4` — синхронизация с соответствующим обратным ходом кадровой развертки монитора.
- `Flags` — флаги, определяющие режим работы метода:
  - `0` — обычный вывод на экран;
  - `DXGI_PRESENT_TEST` — тест состояния цепочки переключений, без вывода изображения (используется для получения информации о том, не заслонено ли окно программы каким-либо другим окном).

IDXGISwapChain::ResizeBuffers

Изменяет размер вторичных буферов цепочки переключений, их формат и количество. Вызывается при изменении размеров окна приложения. Прототип метода:

```
HRESULT ResizeBuffers(
  UINT BufferCount,
  UINT Width,
  UINT Height,
  DXGI_FORMAT NewFormat,
  UINT SwapChainFlags
);
```

Параметры:

- `BufferCount` — количество буферов в цепочке переключений (общее количество, включающее первичный буфер и все вторичные);
- `Width` — новая ширина вторичного буфера, если указано значение 0, автоматически принимается равным ширине клиентской области окна, куда производится вывод;

- `Height` — новая высота вторичного буфера, аналогично, если указано значение 0, автоматически принимается равным высоте клиентской области окна, куда производится вывод;
- `NewFormat` — новый формат вторичного буфера;
- `SwapChainFlags` — дополнительные флаги, задающие параметры переключения буферов.

IDXGISwapChain::SetFullscreenState

Устанавливает либо полноэкранный, либо оконный режим для указанного интерфейса `IDXGIOutput`. Прототип метода:

```
HRESULT SetFullscreenState(
 BOOL Fullscreen,
 IDXGIOutput *pTarget
);
```

Параметры:

- `Fullscreen` — значение `TRUE` — для полноэкранного режима, `FALSE` — для оконного;
- `pTarget` — указатель на интерфейс `IDXGIOutput`, отвечающий за вывод изображения на монитор. Грубо говоря, он определяет, изображение на каком из подключенных мониторов переключить на полный экран.

Функции

Теперь мы опишем наиболее часто употребляемые функции.

D3D10CreateBlob

Создает буфер для произвольных данных. Прототип функции:

```
HRESULT D3D10CreateBlob(
  SIZE_T NumBytes,
  LPD3D10BLOB *ppBuffer
);
```

Параметры:

- `NumBytes` — количество выделяемой памяти в байтах;
- `ppBuffer` — адрес указателя, который будет установлен на создаваемый буфер (см. интерфейс `ID3D10Blob`).

D3D10CreateDeviceAndSwapChain

Создает устройство Direct3D 10 и цепочку переключений. Прототип функции:

```
HRESULT D3D10CreateDeviceAndSwapChain(
  IDXGIAdapter *pAdapter,
  D3D10_DRIVER_TYPE DriverType,
  HMODULE Software,
  UINT Flags,
  UINT SDKVersion,
  DXGI_SWAP_CHAIN_DESC *pSwapChainDesc,
  IDXGISwapChain **ppSwapChain,
  ID3D10Device **ppDevice
)
```

Параметры:

- `pAdapter` — указатель на интерфейс `IDXGIAdapter`, который определяет, к какому видеоадаптеру в системе будет привязано создаваемое устройство, если передается параметр `NULL`, используется видеоадаптер по умолчанию.
- `DriverType` — тип драйвера создаваемого устройства, если видеоадаптер поддерживает DirectX 10, можно создать устройство, использующее аппаратное ускорение графики, для чего нужно указывать тип `D3D10_DRIVER_TYPE_HARDWARE`, в противном случае можно создать устройство, использующее программную эмуляцию функций Direct3D 10, для этого следует указать значение `D3D10_DRIVER_TYPE_REFERENCE`.
- `Software` — описатель библиотеки DLL, в которой реализован программный растеризатор, используется только в случае, если указан тип драйвера `D3D10_DRIVER_TYPE_SOFTWARE`, иначе должно указываться значение `NULL` (этот параметр зарезервирован для использования в будущих версиях).
- `Flags` — необязательные флаги создания устройства, регулируют использование дополнительных возможностей API. например, для вывода дополнительной отладочной информации и расширенных возможностей отладки можно указать флаг `D3D10_CREATE_DEVICE_DEBUG`.
- `SDKVersion` — версия SDK. Всегда используйте значение `D3D10_SDK_VERSION`, которое определено в файле d3d10.h.
- `pSwapChainDesc` — указатель на переменную типа `DXGI_SWAP_CHAIN_DESC`, в которой содержатся параметры создаваемой цепочки переключений.
- `ppSwapChain` — адрес указателя на интерфейс `IDXGISwapChain`, этот указатель функция установит на вновь созданную цепочку переключений.
- `ppDevice` — адрес указателя на интерфейс `ID3D10Device`, этот указатель функция установит на вновь созданное устройство.

D3DX10CreateEffectFromFile

Создает эффект из файла. Прототип функции:

```
HRESULT D3DX10CreateEffectFromFile(
  LPCSTR pFileName,
  CONST D3D10_SHADER_MACRO *pDefines,
  ID3D10Include *pInclude,
  LPCSTR pProfile,
  UINT HLSLFlags,
  UINT FXFlags,
  ID3D10Device *pDevice,
  ID3D10EffectPool *pEffectPool,
  ID3DX10ThreadPump *pPump,
  ID3D10Effect **ppEffect,
  ID3D10Blob **ppErrors
  HRESULT *pHResult
)
```

Параметры:

- `pFileName` — имя файла эффектов в символах ASCII;
- `pDefines` — массив с завершающим нулем, содержащий описания макросов шейдера, которые используются в файле эффектов (может иметь значение `NULL`);
- `pInclude` — подключаемый файл, необходимый для успешной компиляции (может иметь значение `NULL`);
- `pProfile` — профиль шейдера, определяющий модель шейдеров, которая будет использоваться при компиляции файла эффектов (в Windows Vista всегда используется четвертая модель: `fx_4_0`);
- `HLSLFlags` — параметры (флаги) компиляции, относящиеся к шейдерам и типам данных;
- `FXFlags` — параметры (флаги) компиляции, относящиеся к эффектам;
- `pDevice` — указатель на устройство Direct3D 10, к которому будет относиться создаваемый ресурс эффекта;
- `pEffectPool` — указатель на пул эффектов, который нужен для совместного использования переменных несколькими эффектами;
- `pPump` — указатель на интерфейс обработки потоков для организации асинхронной работы, значение `NULL` указывает на отсутствие асинхронной работы;
- `ppEffect` — адрес указателя на создаваемый эффект;

- `ppErrors` — адрес указателя на область памяти, в которой содержится описание ошибок, возникших при компиляции, если, конечно, они возникли;
- `pHResult` — указатель на переменную, в которую возвращается код завершения функции.

D3DX10CreateFont

Создает шрифт и шритовой объект устройства. Прототип функции:

```
HRESULT D3DX10CreateFont(
  ID3D10Device *pDevice,
  INT Height,
  UINT Width,
  UINT Weight,
  UINT MipLevels,
  BOOL Italic,
  UINT CharSet,
  UINT OutputPrecision,
  UINT Quality,
  UINT PitchAndFamily,
  LPCSTR pFaceName,
  LPD3DX10FONT *ppFont
)
```

Параметры:

- `pDevice` — указатель на интерфейс `ID3D10Device`, на устройство, с которым будет связан шрифтовой объект;
- `Height` и `Width` — высота и ширина символов шрифта в логических единицах (в простых случаях логическая единица соответствует пикселу экрана);
- `Weight` — толщина линий символа шрифта;
- `MipLevels` — количество уровней мипмаппинга;
- `Italic` — булевский флаг, указывающий на наклонное начертание символов (italic);
- `CharSet` — код символьного набора шрифта (некоторые из возможных значений приведены в табл. П1.2).
- `OutputPrecision` — флаг, определяющий, каким образом система Windows согласовывает размеры и другие характеристики существующих шрифтов с параметрами, указанным в функции.

- `Quality` — флаг, определяющий, как система Windows должна согласовывать требуемый и используемый шрифт. Применяется только к растровым шрифтам и не должно использоваться для шрифтов TrueType.
- `PitchAndFamily` — информация о шаге символов в строке и о принадлежности шрифта к тому или иному семейству (то есть, например, фиксирована ли ширина символов, или она варьируется). Подробнее см. табл. П1.3 и П1.4).
- `ppFont` — возвращает указатель на интерфейс `ID3DX10Font`, представляющий собой созданный шрифтовой объект.

Таблица П1.2. *Некоторые значения, определяющие код символьного набора шрифта*

Значение	Описание
ANSI_CHARSET	Набор символов ANSI
DEFAULT_CHARSET	Набор символов по умолчанию
RUSSIAN_CHARSET	Набор символов русского алфавита
SYMBOL_CHARSET	Шрифт состоит не из букв, а из картинок

Таблица П1.3. *Значения флага, определяющего шаг символов шрифта*

Значение	Описание
DEFAULT_PITCH	Использовать шаг по умолчанию
FIXED_PITCH	Использовать фиксированный шаг
VARIABLE_PITCH	Использовать переменный шаг

Таблица П1.4. *Значения флага, определяющего семейство шрифта*

Значение	Описание
FF_DECORATIVE	Декоративный шрифт
FF_DONTCARE	По умолчанию
FF_MODERN	Шрифт с постоянной шириной штрихов, с засечкам или без. Пример: Pica, Elite и Courier New
FF_ROMAN	Шрифт с переменной толщиной штрихов и с засечками. Пример MS Serif
FF_SCRIPT	Шрифт, имитирующий рукописный текст. Пример: шрифты Script и Cursive
FF_SWISS	Шрифт с переменной толщиной штрихов и без засечек. Пример: шрифт MS Sans Serif

D3DX10CreateShaderResourceViewFromFile

Создает представление данных как ресурса шейдера. Прототип функции:

```
HRESULT D3DX10CreateShaderResourceViewFromFile(
  ID3D10Device *pDevice,
  LPCSTR pSrcFile,
  D3DX10_IMAGE_LOAD_INFO *pLoadInfo,
  ID3DX10ThreadPump* pPump,
  ID3D10ShaderResourceView** ppShaderResourceView,
  HRESULT* pHResult
)
```

Параметры:

`pDevice` — указатель на интерфейс устройства Direct3D 10;

`pSrcFile` — указатель на строку с завершающим нулем, в которой содержится имя нужного файла;

`pLoadInfo` — указатель на структуру типа `D3DX10_IMAGE_LOAD_INFO` с информацией о параметрах загрузки текстуры;

`pPump` — указатель на интерфейс конвейера обработки потоков, используется, если необходимо асинхронное выполнение команд;

`ppShaderResourceView` — адрес указателя на представление данных ресурса шейдера;

`pHResult` — указатель на возвращенное значение. Можно указывать значение `NULL`. В случае если `pPump` не `NULL`, параметр `pHResult` должен указывать на корректную область памяти до тех пор, пока асинхронное выполнение команд не завершится.

D3DX10CreateSprite

Создает спрайт для вывода на экран двухмерной текстуры. Прототип функции:

```
HRESULT D3DX10CreateSprite(
  ID3D10Device *pDevice,
  UINT cDeviceBufferSize,
  LPD3DX10SPRITE *ppSprite
)
```

Параметры:

- `pDevice` — указатель на интерфейс устройства Direct3D 10, которое будет рисовать спрайт;

- `cDeviceBufferSize` — количество отображаемых спрайтов, максимальная величина 4096;
- `ppSprite` — адрес указателя на интерфейс спрайта, этот указатель будет установлен на созданный спрайт.

D3DX10GetImageInfoFromFile

Считывает информацию об изображении, хранящемся в указанном файле. Прототип функции:

```
HRESULT D3DX10GetImageInfoFromFile(
  LPCTSTR pSrcFile,
  ID3DX10ThreadPump *pPump,
  D3DX10_IMAGE_INFO *pSrcInfo,
  HRESULT *pHResult
);
```

Параметры:

- `pSrcFile` — имя файла с изображением, информацию о котором требуется считать;
- `pPump` — указатель на интерфейс конвейера обработки потоков, используется, если необходимо асинхронное выполнение команд;
- `pSrcInfo` — указатель на структуру типа D3DX10_IMAGE_INFO, в которую заносятся данные об изображении из файла;
- `pHResult` — указатель на возвращенное значение. Можно указывать значение `NULL`. В случае если `pPump` не `NULL`, параметр `pHResult` должен указывать на корректную область памяти до тех пор, пока асинхронное выполнение команд не завершится.

> ***Примечание***
>
> Функция поддерживает работу со строками как с символами ANSI, так и с символами Unicode.

D3DXMatrixIdentity

Создает единичную матрицу. Прототип функции:

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

Параметр:

- `pOut` — указатель на структуру типа `D3DXMATRIX`, которая устанавливается равной единичной матрице.

D3DXMatrixLookAtLH

Создает матрицу вида с использованием левосторонней системы координат с камерой, направленной в заданную точку. Прототип функции:

```
D3DXMATRIX * D3DXMatrixLookAtLH(
  D3DXMATRIX *pOut,
  CONST D3DXVECTOR3 *pEye,
  CONST D3DXVECTOR3 *pAt,
  CONST D3DXVECTOR3 *pUp
);
```

Параметры:

- `pOut` — указатель на структуру типа `D3DXMATRIX`, над которой выполняется операция;
- `pEye` — указатель на структуру типа `D3DXVECTOR3`, которая определяет точку положения камеры;
- `pAt` — указатель на структуру типа `D3DXVECTOR3`, которая задает точку, на которую направлена камера;
- `pUp` — указатель на структуру типа `D3DXVECTOR3`, которая задает направление вверх, обычно это вектор (0, 1, 0).

D3DXMatrixOrthoLH

Создает матрицу ортографической проекции с использованием левосторонней системы координат. Прототип функции:

```
D3DXMATRIX * D3DXMatrixOrthoLH(
  D3DXMATRIX *pOut,
  FLOAT w,
  FLOAT h,
  FLOAT zn,
  FLOAT zf
);
```

Параметры:

- `pOut` — указатель на результирующую структуру типа `D3DXMATRIX`;
- `w` и `h` — ширина и высота объема видимости;
- `zn` — минимальное значение координаты Z, определяющее ближнюю плоскость отсечения;
- `zf` — минимальное значение координаты Y, определяющее дальнюю плоскость отсечения.

Примечание

Значения всех параметров представляют собой расстояния в пространстве координат камеры.

D3DXMatrixPerspectiveFovLH

Создает матрицу перспективной проекции с использованием левосторонней системы координат для заданного поля зрения. Прототип функции:

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

Параметры:

- `pOut` — указатель на результирующую матрицу;
- `fovy` — угол, ограничивающий поле зрения в направлении оси Y, указывается в радианах. Обычно используется значение `D3DX_PI/4`;
- `Aspect` — соотношение сторон окна либо экрана, если изображение выводится в полноэкранном режиме;
- `zn` — координата Z ближней плоскости отсечения;
- `zf` — координата Z дальней плоскости отсечения, объекты, находящиеся за ней, считаются расположенными за горизонтом и не отображаются.

D3DXMatrixRotationX

Создает матрицу вращения вокруг оси X на заданный угол. Прототип функции:

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

Параметры:

- `pOut` — указатель на результирующую матрицу;
- `Angle` — угол поворота в радианах.

D3DXMatrixRotationY

Создает матрицу вращения вокруг оси Y на заданный угол. Прототип функции:

```
D3DXMATRIX * D3DXMatrixRotationY(
  D3DXMATRIX *pOut,
  FLOAT Angle
);
```

Параметры:

- `pOut` — указатель на результирующую матрицу;
- `Angle` — угол поворота в радианах.

D3DXMatrixRotationZ

Создает матрицу вращения вокруг оси Z на заданный угол. Прототип функции:

```
D3DXMATRIX * D3DXMatrixRotationZ(
  D3DXMATRIX *pOut,
  FLOAT Angle
);
```

Параметры:

- `pOut` — указатель на результирующую матрицу;
- `Angle` — угол поворота в радианах.

D3DXMatrixScaling

Создает матрицу масштабирования вдоль осей X, Y и Z. Прототип функции:

```
D3DXMATRIX * D3DXMatrixScaling(
  D3DXMATRIX *pOut,
  FLOAT sx,
  FLOAT sy,
  FLOAT sz
);
```

Параметры:

- `pOut` — указатель на результирующую матрицу;
- `sx` — коэффициент масштабирования по оси X;
- `sy` — коэффициент масштабирования по оси Y;
- `sz` — коэффициент масштабирования по оси Z.

D3DXMatrixTranslation

Создает матрицу перемещения. Прототип функции:

```
D3DXMATRIX* D3DXMatrixTranslation(
  D3DXMATRIX *pOut,
  FLOAT x,
  FLOAT y,
  FLOAT z
);
```

Параметры:

- `pOut` — указатель на матрицу, которая станет матрицей перемещения;
- `x` — составляющая перемещения вдоль оси X;
- `y` — составляющая перемещения вдоль оси Y;
- `z` — составляющая перемещения вдоль оси Z.

Структуры

В этом разделе приведены описания структур, наиболее часто используемых при программировании графики с применением Direct3D 10.

D3D10_BUFFER_DESC

Структура описывает параметры буферного ресурса. Определяется она следующим образом:

```
typedef struct D3D10_BUFFER_DESC {
    UINT ByteWidth;
    D3D10_USAGE Usage;
    UINT BindFlags;
    UINT CPUAccessFlags;
    UINT MiscFlags;
} D3D10_BUFFER_DESC;
```

Значение полей:

- `ByteWidth` — размер буфера в байтах;
- `Usage` — режим использования буфера, показывающий, как требуется производить чтение из буфера и запись в него, ключевым фактором является частота обновления данных в буфере (обычно используется флаг `D3D10_USAGE_DEFAULT`);

- `BindFlags` — флаг, определяющий, в каком качестве буфер связывается с графическим конвейером, поле может содержать несколько объединенных с помощью операции OR флагов;
- `CPUAccessFlags` — флаг, определяющий, какие виды доступа разрешаются для центрального процессора, может содержать несколько объединенных с помощью операции OR флагов (если доступ для центрального процессора не требуется, устанавливается значение 0);
- `MiscFlags` — флаг, определяющий прочие, редко используемые свойства буфера, поле также может содержать несколько объединенных с помощью операции OR флагов. Если не требуется использование ни одного из флагов, устанавливается значение 0.

D3D10_DEPTH_STENCIL_VIEW_DESC

Определяет подресурсы текстуры, к которым имеется доступ при использовании представления данных как шаблонного буфера глубины. Определение структуры выглядит так:

```
typedef struct D3D10_DEPTH_STENCIL_VIEW_DESC {
    DXGI_FORMAT Format;
    D3D10_DSV_DIMENSION ViewDimension;
    union {
        D3D10_TEX1D_DSV Texture1D;
        D3D10_TEX1D_ARRAY_DSV Texture1DArray;
        D3D10_TEX2D_DSV Texture2D;
        D3D10_TEX2D_ARRAY_DSV Texture2DArray;
        D3D10_TEX2DMS_DSV Texture2DMS;
        D3D10_TEX2DMS_ARRAY_DSV Texture2DMSArray;
    };
} D3D10_DEPTH_STENCIL_VIEW_DESC;
```

Поля имеют следующий смысл.

- `Format` — формат данных ресурса. Для представления данных как шаблонного буфера глубины допустимы следующие форматы данных:
  - `DXGI_FORMAT_D16_UNORM`;
  - `DXGI_FORMAT_D24_UNORM_S8_UINT`;
  - `DXGI_FORMAT_D32_FLOAT`;
  - `DXGI_FORMAT_D32_FLOAT_S8X24_UINT`;
  - `DXGI_FORMAT_UNKNOWN`.

Если указан формат DXGI_FORMAT_UNKNOWN, используется формат родительского ресурса.

- D3D10_DSV_DIMENSION — тип ресурса, определяет, каким образом будет производиться доступ к ресурсу. Значение этого поля хранится в структуре в виде объединения (union).
- Texture1D — определяет подресурс как одномерную текстуру.
- Texture1DArray — определяет подресурс как массив одномерных текстур.
- Texture2D — определяет подресурс как двухмерную текстуру.
- Texture2DArray — определяет подресурс как массив двухмерных текстур.
- Texture2DMS — не используется (двухмерная текстура с мультисэмлированием содержит единственный подресурс).
- Texture2DMSArray — не используется (двухмерная текстура с мультисэмлированием содержит единственный подресурс на текстуру).

Примечание

Описание представления данных как шаблонного буфера глубины требуется указывать при вызове метода ID3D10Device::CreateDepthStencilView().

D3D10_INPUT_ELEMENT_DESC

Описывает одиночный элемент данных, которые подаются на вход графического конвейера. Определение структуры:

```
typedef struct D3D10_INPUT_ELEMENT_DESC {
    LPCSTR SemanticName;
    UINT SemanticIndex;
    DXGI_FORMAT Format;
    UINT InputSlot;
    UINT AlignedByteOffset;
    D3D10_INPUT_CLASSIFICATION InputSlotClass;
    UINT InstanceDataStepRate;
} D3D10_INPUT_ELEMENT_DESC;
```

Поля имеют следующий смысл.

- SemanticName — семантика HLSL, которая связывает данный элемент с входным параметром шейдера с такой же семантикой.
- SemanticIndex — дополнительный семантический индекс элемента, используется только в случаях, когда необходимо передавать несколько элементов с одинаковой семантикой. Этот индекс меняет семантическое имя, добавляя к нему указанный индекс.

- `Format` — тип данных (формат) элемента.
- `InputSlot` — номер входного канала конвейера, допустимые значения: 0 — 15.
- `AlignedByteOffset` — необязательный параметр, который представляет собой смещение описываемого элемента в байтах от начала данных вершины. Информация о вершине для процессора видеокарты представляет собой обычный фрагмент памяти, а смещение показывает, с какого байта начинать считывание элемента массива. Чтобы не высчитывать смещение самому, а просто указать, что элемент следует сразу за предыдущим, можно использовать константу `D3D10_APPEND_ALIGNED_ELEMENT`.
- `InputSlotClass` — определяет тип данных, передаваемых по входному каналу.
- `InstanceDataStepRate` — используется при создании экземпляров (instancing), в случае, если передаются данные об одной вершине, это поле должно содержать нулевое значение.

D3D10_PASS_DESC

Описывает проход визуализации. Определение структуры:

```
typedef struct D3D10_PASS_DESC {
    LPCSTR Name;
    UINT Annotations;
    BYTE *pIAInputSignature;
    SIZE_T IAInputSignatureSize;
    UINT StencilRef;
    UINT SampleMask;
    FLOAT BlendFactor[4];
} D3D10_PASS_DESC;
```

Назначение полей:

- `Name` — строка, содержащая название прохода, в противном случае — `NULL`;
- `Annotations` — количество аннотаций;
- `pIAInputSignature` — сигнатура входных данных вершинного шейдера, в противном случае — `NULL`;
- `IAInputSignatureSize` — размер сигнатуры в байтах;
- `StencilRef` — контрольное значение для шаблона, используемое в режиме шаблонного буфера глубины;

- `SampleMask` — маска образца для режима смешивания текстур;
- `BlendFactor` — множители смешивания для режима смешивания текстур.

> **Примечание**
>
> Техника отображения эффекта может содержать 1 или более проходов. Чтобы получить описание прохода, можно вызвать метод `ID3D10EffectPass::GetDesc()`.

D3D10_SUBRESOURCE_DATA

Определяет данные для инициализации подресурсов. Определение структуры:

```
typedef struct D3D10_SUBRESOURCE_DATA {
    const void *pSysMem;
    UINT SysMemPitch;
    UINT SysMemSlicePitch;
} D3D10_SUBRESOURCE_DATA;
```

Назначение полей:

- `pSysMem` — указатель на данные для инициализации;
- `SysMemPitch` — шаг в памяти (в байтах), который используется только для данных двухмерных и трехмерных текстур, так как для ресурсов другого типа он не имеет смысла;
- `SysMemSlicePitch` — размер одного уровня глубины (в байтах). Это поле используется только для данных трехмерных текстур, так как оно не имеет смысла для ресурсов других типов.

> **Примечание**
>
> Эта структура используется при создании буфера данных с помощью метода `ID3D10Device::CreateBuffer()` и при создании текстур с помощью методов `ID3D10Device::CreateTexture1D()`, `ID3D10Device::CreateTexture2D()`, `ID3D10Device::CreateTexture3D()`.

D3D10_TEXTURE2D_DESC

Описывает двухмерную текстуру. Определение структуры:

```
typedef struct D3D10_TEXTURE2D_DESC {
    UINT Width;
    UINT Height;
    UINT MipLevels;
    UINT ArraySize;
```

```
    DXGI_FORMAT Format;
    DXGI_SAMPLE_DESC SampleDesc;
    D3D10_USAGE Usage;
    UINT BindFlags;
    UINT CPUAccessFlags;
    UINT MiscFlags;
} D3D10_TEXTURE2D_DESC;
```

Назначение полей:

- `Width` и `Height` — ширина и высота текстуры (в текселах);
- `MipLevels` — количество уровней мипмаппинга, для текстур с мультисэмплингом указывается значение 1, значение 0 генерирует полный набор текстур для соответствующих уровней;
- `ArraySize` — количество текстур в текстурном массиве;
- `Format` — формат текстуры;
- `SampleDesc` — структура, описывающая параметры мультисэмплинга для данной текстуры.;
- `Usage` — значение, определяющее, в каком режиме будет производиться запись и чтение данных из текстуры, в основном используется значение `D3D10_USAGE_DEFAULT`;
- `BindFlags` — флаги для связи со стадиями графического конвейера, флаги можно комбинировать при помощи логического оператора OR;
- `CPUAccessFlags` — флаги, определяющие, в каких режимах разрешается доступ центральному процессору к данным текстуры, нулевое значение означает, что доступ для центрального процессора не требуется;
- `MiscFlags` — флаги, определяющие другие, более редко используемые параметры. Флаги можно комбинировать при помощи логического оператора OR.

> ***Примечание***
>
> Эта структура используется при создании текстуры с помощью метода `ID3D10Device::CreateTexture2D()`.

D3D10_VIEWPORT

Задает размеры области отображения. Определение структуры:

```
typedef struct D3D10_VIEWPORT {
    INT TopLeftX;
    INT TopLeftY;
```

```
    UINT Width;
    UINT Height;
    FLOAT MinDepth;
    FLOAT MaxDepth;
} D3D10_VIEWPORT;
```

Назначение полей:

- `TopLeftX` — координата X левой стороны области отображения, может принимать значения от –16384 до 16383;
- `TopLeftY` — координата Y верхней стороны области отображения, может принимать значения от –16384 до 16383;
- `Width` и `Height` — ширина и высота области отображения, могут принимать значения от –16384 до 16383;
- `MinDepth` и `MaxDepth` — минимальная и максимальная глубина области отображения, может принимать значения от 0 до 1.

D3DX10_IMAGE_INFO

Содержит данные о первоначальном содержимом файла изображения. Определение структуры:

```
typedef struct D3DX10_IMAGE_INFO {
    UINT Width;
    UINT Height;
    UINT Depth;
    UINT ArraySize;
    UINT MipLevels;
    UINT MiscFlags;
    DXGI_FORMAT Format;
    D3D10_RESOURCE_DIMENSION ResourceDimension;
    D3DX10_IMAGE_FILE_FORMAT ImageFileFormat;
} D3DX10_IMAGE_INFO, *LPD3DX10_IMAGE_INFO;
```

Назначение полей:

- `Width`, `Height` и `Depth` — ширина, высота и глубина исходного изображения в пикселах;
- `ArraySize` — размер изображения в байтах;
- `MipLevels` — количество уровней мипмаппинга в исходном изображении;
- `MiscFlags` — прочие свойства ресурса;
- `Format` — значение формата типа `D3D10FORMAT`, наиболее близко описывающее данные исходного изображения;

- `ResourceDimension` — представляет собой тип сохраненной в файле текстуры;
- `ImageFileFormat` — формат файла изображения.

D3DX10_IMAGE_LOAD_INFO

Предоставляет информацию о способе загрузки текстуры. Определение структуры:

```
typedef struct D3DX10_IMAGE_LOAD_INFO {
  UINT Width;
  UINT Height;
  UINT Depth;
  UINT FirstMipLevel;
  UINT MipLevels;
  D3D10_USAGE Usage;
  UINT BindFlags;
  UINT CpuAccessFlags;
  UINT MiscFlags;
  DXGI_FORMAT Format;
  UINT Filter;
  UINT MipFilter;
  D3DX10_IMAGE_INFO* pSrcInfo;
} D3DX10_IMAGE_LOAD_INFO, *LPD3DX10_IMAGE_LOAD_INFO;
```

Назначение полей:

- `Width` и `Height` — требуемые ширина и высота текстуры, если фактические ширина и высота не совпадают с заданными значениями, текстура будет отмасштабирована, чтобы соответствовать указанным размерам;
- `Depth` — глубина текстуры, имеет смысл только для объемных текстур;
- `FirstMipLevel` — уровень для мипмаппинга с наибольшим разрешением, в случае, если установлено значение больше 0, после загрузки текстуры уровень `FirstMapLevel` будет назначен на уровень 0;
- `MipLevels` — максимальное количество уровней для мипмаппинга, которое будет иметь текстура;
- `Usage` — поле, определяющее режим использования текстурного ресурса;
- `BindFlags` — флаги, определяющие стадии графического конвейера, с которыми будет разрешено связывать текстуру;
- `CpuAccessFlags` — флаги, определяющие, какие разрешения на доступ к ресурсу будет иметь центральный процессор;

- `MiscFlags` — флаги, определяющие прочие, реже используемые свойства ресурса;
- `Format` — формат, который текстура будет иметь после загрузки;
- `Filter` — флаги, задающие фильтрацию текстуры;
- `MipFilter` — флаг, задающий фильтрацию для уровней мипмаппинга;
- `pSrcInfo` — указатель на структуру типа `D3DX10_IMAGE_INFO`, в которой содержатся данные о первоначальном изображении (об изображении, которое хранится в файле).

D3DX10_SPRITE

Задает положение, текстуру и цветовую информацию спрайта. Определение структуры:

```
typedef struct D3DX10_SPRITE {
  D3DXMATRIX matWorld;
  D3DXVECTOR2 TexCoord;
  D3DXVECTOR2 TexSize;
  D3DXCOLOR ColorModulate;
  ID3D10ShaderResourceView *pTexture;
  UINT TextureIndex;
} D3DX10_SPRITE;
```

Назначение полей:

- `matWorld` — мировая матрица спрайта, которая определяет его положение и ориентацию в пространстве мировых координат;
- `TexCoord` — вектор, направленный из верхнего левого угла текстуры к верхнему левому углу спрайта, выраженный в текстурных координатах (фактически определяет смещение текстуры относительно верхнего левого угла спрайта);
- `TexSize` — вектор, направленный из верхнего левого угла спрайта в нижний правый угол спрайта, выраженный в текстурных координатах (фактически определяет, сколько раз будет повторяться текстура вдоль вертикальной и горизонтальной осей спрайта);
- `ColorModulate` — цвет, на который необходимо умножить цвет пиксела текстуры перед его визуализацией;
- `pTexture` — указатель на представление данных как ресурса шейдера, содержащее текстуру спрайта;
- `TextureIndex` — индекс текстуры в массиве текстур, если текстура не содержится в массиве, здесь должно находиться нулевое значение.

D3DXVECTOR2

Определяет вектор в двухмерном пространстве. Структура:

```
typedef struct D3DXVECTOR2 {
    FLOAT x;
    FLOAT y;
} D3DXVECTOR2, *LPD3DXVECTOR2;
```

Назначение полей:

- `x` — координата X;
- `y` — координата Y.

D3DXVECTOR3

Определяет вектор в трехмерном пространстве. Структура:

```
typedef struct D3DXVECTOR3 {
    FLOAT x;
    FLOAT y;
    FLOAT z;
} D3DXVECTOR3, *LPD3DXVECTOR3;
```

Назначение полей:

- `x` — координата X;
- `y` — координата Y;
- `z` — координата Z.

D3DXVECTOR4

Определяет вектор в четырехмерном пространстве. Структура:

```
typedef struct D3DXVECTOR3 {
    FLOAT x;
    FLOAT y;
    FLOAT z;
    FLOAT w;
} D3DXVECTOR3, *LPD3DXVECTOR3;
```

Назначение полей:

- x — координата X;
- y — координата Y;
- z — координата Z;
- w — координата W.

DXGI_SWAP_CHAIN_DESC

Структура содержит описание параметров цепочки переключений. Определение структуры:

```
typedef struct DXGI_SWAP_CHAIN_DESC {
    DXGI_MODE_DESC BufferDesc;
    DXGI_SAMPLE_DESC SampleDesc;
    DXGI_USAGE BufferUsage;
    UINT BufferCount;
    HWND OutputWindow;
    BOOL Windowed;
    DXGI_SWAP_EFFECT SwapEffect;
    UINT Flags;
} DXGI_SWAP_CHAIN_DESC;
```

Параметры имеют следующий смысл.

- `BufferDesc` — описывает режим работы дисплея.
- `SampleDesc` — назначает параметры мультисэмплинга.
- `BufferUsage` — определяет режим использования вторичного буфера и доступ к нему центрального процессора. При использовании Direct3D 10 вторичный буфер можно использовать в качестве входных данных для шейдера либо в качестве буфера визуализации для вывода данных. Эти режимы могут сочетаться с любыми флагами для доступа центрального процессора.
- `BufferCount` — задает количество буферов в цепочке переключений.
- `OutputWindow` — содержит описатель окна, в которое осуществляется вывод. Если уже существует родительский интерфейс и имеется связанное с ним окно, то в качестве этого параметра можно передать значение `NULL`, и в этом случае будет использоваться связанное окно.
- `Windowed` — содержит значение `TRUE`, если вывод осуществляется в оконном режиме, и `FALSE` — если в полноэкранном.
- `SwapEffect` — определяет влияние работы метода `IDXGISwapChain::Present()` на содержимое буфера.
- `Flags` — определяет работу цепочки переключений посредством набора флагов.

RECT

Описывает прямоугольную область. Определение структуры:

```
typedef struct RECT {
  LONG left;
  LONG top;
  LONG right;
  LONG bottom;
}
```

Назначение полей:

- `left` — координата X верхнего левого угла прямоугольника;
- `top` — координата Y верхнего левого угла прямоугольника;
- `right` — координата X нижнего правого угла прямоугольника;
- `bottom` — координата Y нижнего правого угла прямоугольника.

Приложение 2

Справочник функций DXUT

Все функции в справочнике разбиты на пять разделов, так же, как это сделано в *Приложении 1*:

- инициализация DXUT;
- установка функций обратного вызова;
- выполнение стандартных задач;
- получение сведений об установках DXUT;
- структуры.

Инициализация DXUT

DXUTCreateDevice

Создает устройство Direct3D 9 либо Direct3D 10. Прототип функции:

```
HRESULT DXUTCreateDevice(
  bool bWindowed,
  INT nSuggestedWidth,
  INT nSuggestedHeight
);
```

Параметры:

- `bWindowed` — флаг, указывающий, в каком режиме запускается приложение. Приложение запустится в оконном режиме, если передано значение `TRUE`, и в полноэкранном, если передано значение `FALSE`. По умолчанию установлено значение `TRUE`.
- `nSuggestedWidth` и `nSuggestedHeight` — требуемые ширина и высота вторичного буфера в пикселах. Указанные значения могут быть скорректированы

исходя из ограничений операционной системы и аппаратных возможностей. Значения по умолчанию для обоих параметров — 0.

Если оба параметра `nSuggestedWidth` и `nSuggestedHeight` равны нулю, будут использованы соответствующие размеры клиентской области окна.

DXUTCreateWindow

Создает окно для приложения DXUT. Прототип функции:

```
HRESULT DXUTCreateWindow(
  CONST const WCHAR *strWindowTitle,
  HINSTANCE hInstance,
  HICON hIcon,
  HMENU hMenu,
  INT x,
  INT y
);
```

Параметры:

- `strWindowTitle` — указатель на строку, содержащую заголовок окна. Строка должна использовать символы Unicode, по умолчанию используется текст "Direct3D Window".
- `hInstance` — описатель экземпляра приложения либо значение `NULL` для получения описателя текущего модуля. Значение по умолчанию — `NULL`.
- `hIcon` — описатель значка приложения либо значение `NULL`, если нужно использовать первый значок, включенный в исполняемый файл. По умолчанию установлено значение `NULL`.
- `hMenu` — описатель ресурса меню либо значение `NULL`, чтобы указать, что меню отсутствует. Значение по умолчанию — `NULL`.
- `x` и `y` — горизонтальная и вертикальная координаты верхнего левого угла окна в системе координат экрана. Значение `CW_USEDEFAULT` позволяет Windows самой определить соответствующее положение. По умолчанию используется `CW_USEDEFAULT`.

DXUTInit

Инициализирует каркас DXUT. Прототип функции:

```
HRESULT DXUTInit(
  BOOL bParseCommandLine,
  BOOL bShowMsgBoxOnError,
```

```
  WCHAR *strExtraCommandLineParams,
  bool bThreadSafeDXUT
);
```

Параметры:

- `bParseCommandLine` — булевский флаг, указывающий, следует ли производить анализ параметров командной строки (см. табл. П2.1). По умолчанию разбор командной строки производится.
- `bShowMsgBoxOnError` — параметр, который отвечает за вывод сообщений об ошибках. По умолчанию сообщения об ошибках выводятся.
- `strExtraCommandLineParams` — указатель на дополнительные параметры командной строки. По умолчанию здесь стоит значение `NULL`.
- `bThreadSafeDXUT` — булевский флаг, определяющий, поддерживают ли объекты DXUT многопоточность. По умолчанию стоит значение `FALSE`.

Таблица П2.1. *Параметры командной строки*

Параметр	Значение
-adapter:#	Использовать адаптер # (не работает при отсутствии такого адаптера)
-windowed	Запустить приложение в оконном режиме
-fullscreen	Запустить приложение в полноэкранном режиме
-forcehal	Принудительно использовать HAL (аппаратное ускорение), не работает, если таковое не поддерживается
-forceref	Принудительно использовать REF (программную эмуляцию графического конвейера), не работает при отсутствии возможности использования такого устройства
-forcepurehwvp	(Применимо только к Direct3D 9.) Принудительно использовать исключительно HWVP (аппаратную обработку вершин), не работает, если этот режим не поддерживается видеоадаптером
-forcehwvp	(Применимо только к Direct3D 9.) Принудительно использовать HWVP (аппаратную обработку вершин). Не работает, если этот режим не поддерживается видеоадаптером
-forceswvp	(Применимо только к Direct3D 9.) Принудительно использовать SWVP (программную обработку вершин), не работает, если этот режим не поддерживается видеоадаптером
-forcevsync:#	Устанавливает синхронизацию вывода изображения: значение 0 отключает синхронизацию, любое другое значение — включает

Таблица П2.1 *(окончание)*

Параметр	Значение
-width:#	Принудительно устанавливает ширину окна в # пикселов, для полноэкранного режима будет подобран ближайший из поддерживаемых режимов
-height:#	Принудительно устанавливает высоту окна в # пикселов, для полноэкранного режима будет подобран ближайший из поддерживаемых режимов
-startx:#	Указывает, что в оконном режиме нужно поместить окно в позицию с указанной координатой по оси X
-starty:#	Указывает, что в оконном режиме нужно поместить окно в позицию с указанной координатой по оси Y
-constantframetime:#	Заставляет приложение выделять для прорисовки кадра фиксированный интервал времени, который задается в миллисекундах
-quitafterframe:x	Заставляет приложение завершиться после вывода x кадров
-noerrormsgboxes	Отключает вывод сообщений об ошибках средствами каркаса DXUT
-nostats	Отключает вывод данных об устройстве и статистике о количестве кадров в секунду (функции DXUTGetDeviceStats и DXUTGetFrameStats будут возвращать пустые строки)
-automation	Включает возможность навигации по элементам интерфейса при помощи клавиатуры. Используется по умолчанию

DXUTMainLoop

Запускает главный цикл программы DXUT. Прототип функции:

```
HRESULT DXUTMainLoop(
  HACCEL hAccel
);
```

Параметр:

- `hAccel` — указатель на таблицу "горячих клавиш", используемых при обработке сообщений из очереди, либо значение `NULL`, если эта таблица не используется. Значение по умолчанию — `NULL`.

Эта функция запускает цикл обработки сообщений, который будет выполняться, пока работает программа. Функция будет вызывать зарегистрированные функции обратного вызова чтобы сообщить приложению о необходимости пересчитать и перерисовать трехмерную сцену, а также обработать события от устройств ввода.

DXUTRender3DEnvironment

Выполняет прорисовку трехмерной сцены. Прототип функции:

```
VOID DXUTRender3DEnvironment();
```

Параметры: отсутствуют.

Обычно эту функцию вызывать не требуется. Она используется только в том случае, если приложение не вызывает функцию `DXUTMainLoop()`, но нуждается в поддержке DXUT для визуализации сцены.

Установка функций обратного вызова

DXUTSetCallbackD3D10DeviceAcceptable

Устанавливает функцию обратного вызова, сигнализирующую о приемлемости режима видеоадаптера. Прототип функции:

```
VOID DXUTSetCallbackD3D10DeviceAcceptable(
  LPDXUTCALLBACKISD3D10DEVICEACCEPTABLE pCallback,
  void *pUserContext
);
```

Параметры:

- `pCallback` — указатель на функцию обратного вызова;
- `pUserContext` — указатель на определяемое пользователем значение, которое передается функции. Значение по умолчанию — `NULL`.

DXUTSetCallbackD3D10DeviceCreated

Устанавливает функцию, вызываемую после создания устройства Direct3D 10. Прототип функции:

```
VOID DXUTSetCallbackD3D10DeviceCreated(
  LPDXUTCALLBACKD3D10DEVICECREATED pCallback,
  void *pUserContext
);
```

Параметры:

- `pCallback` — указатель на функцию обратного вызова.
- `pUserContext` — указатель на определяемое пользователем значение, которое передается функции. Значение по умолчанию — `NULL`.

DXUTSetCallbackD3D10DeviceDestroyed

Устанавливает функцию, вызываемую после уничтожения устройства Direct3D 10. Прототип функции:

```
VOID DXUTSetCallbackD3D10DeviceDestroyed(
  LPDXUTCALLBACKD3D10DEVICEDESTROYED pCallback,
  void *pUserContext
);
```

Параметры:

- `pCallback` — указатель на функцию обратного вызова. Уничтожение устройства, как правило, является результатом завершения работы программы либо изменения настроек устройства Direct3D 10. Если передано значение `NULL`, DXUT не будет извещать приложение об уничтожении устройства.
- `pUserContext` — указатель на определяемое пользователем значение, которое передается функции. Значение по умолчанию — `NULL`.

DXUTSetCallbackD3D10FrameRender

Устанавливает функцию обратного вызова для визуализации трехмерной сцены. Прототип функции:

```
VOID DXUTSetCallbackD3D10FrameRender(
  LPDXUTCALLBACKD3D10FRAMERENDER pCallback,
  void *pUserContext
);
```

Параметры:

- `pCallback` — указатель на функцию обратного вызова. Если функция установлена, она будет вызываться для прорисовки каждого кадра. Если передано значение `NULL`, DXUT не будет сообщать приложению о необходимости вывода следующего кадра.
- `pUserContext` — указатель на определяемое пользователем значение, которое передается функции. Значение по умолчанию — `NULL`.

DXUTSetCallbackD3D10SwapChainReleasing

Устанавливает функцию для вызова после освобождения цепочки переключений. Прототип функции:

```
HRESULT DXUTSetCallbackD3D10SwapChainReleasing(
  LPDXUTCALLBACKD3D10SWAPCHAINRELEASING pCallback,
  void *pUserContext
);
```

Параметры:

- `pCallback` — указатель на функцию обратного вызова. Если функция установлена, она будет вызываться перед освобождением цепочки переключений. Цепочка переключений освобождается, если произошло изменение размеров окна, либо разрешения экрана. Если передано значение `NULL`, DXUT не будет сообщать приложению о событиях цепочки переключений.
- `pUserContext` — указатель на определяемое пользователем значение, которое передается функции. Значение по умолчанию — `NULL`.

DXUTSetCallbackD3D10SwapChainResized

Устанавливает функцию для вызова после изменения размеров буферов цепочки переключений. Прототип функции:

```
HRESULT DXUTSetCallbackD3D10SwapChainResized(
  LPDXUTCALLBACKD3D10SWAPCHAINRESIZED pCallback,
  void *pUserContext
);
```

Параметры:

- `pCallback` — указатель на функцию обратного вызова. Если функция установлена, она будет вызываться перед тем, как цепочка переключений создана и изменила размеры буферов. Цепочка переключений меняет размер вторичных буферов, если произошло изменение размеров окна либо разрешения экрана. Если передано значение `NULL`, DXUT не будет сообщать приложению о событиях цепочки переключений.
- `pUserContext` — указатель на определяемое пользователем значение, которое передается функции. Значение по умолчанию — `NULL`.

> ***Примечание***
>
> Ресурсы, выделенные в функции обратного вызова, должны быть освобождены в функции, установленной вызовом `DXUTSetCallbackD3D10SwapChainReleasing`.

DXUTSetCallbackDeviceChanging

Устанавливает функцию, которая позволит приложению поменять настройки устройства Direct3D перед его созданием. Прототип функции:

```
VOID DXUTSetCallbackDeviceChanging(
  LPDXUTCALLBACKMODIFYDEVICESETTINGS pCallbackModifyDeviceSettings,
  void *pUserContext
);
```

Параметры:

- `pCallbackModifyDeviceSettings` — указатель на функцию обратного вызова. Если функция установлена, она будет вызываться перед созданием устройства Direct3D. Если передано значение `NULL`, DXUT не будет сообщать приложению о смене устройств Direct3D.
- `pUserContext` — указатель на определяемое пользователем значение, которое передается функции. Значение по умолчанию — `NULL`.

DXUTSetCallbackFrameMove

Устанавливает функцию для расчета очередного кадра трехмерной сцены. Прототип функции:

```
VOID DXUTSetCallbackFrameMove(
  LPDXUTCALLBACKFRAMEMOVE pCallbackFrameMove,
  void *pUserContext
);
```

Параметры:

- `pCallbackFrameMove` — указатель на функцию обратного вызова. Если функция установлена, она будет вызываться перед прорисовкой каждого кадра для осуществления необходимых изменений в трехмерной сцене. Если передано значение `NULL`, DXUT не будет сообщать приложению о необходимости обновлений для новых кадров.
- `pUserContext` — указатель на определяемое пользователем значение, которое передается функции. Значение по умолчанию — `NULL`.

DXUTSetCallbackMsgProc

Устанавливает функцию обратного вызова для обработки сообщений. Прототип функции:

```
VOID DXUTSetCallbackMsgProc(
  LPDXUTCALLBACKMSGPROC pCallbackMsgProc,
  void *pUserContext
);
```

Параметры:

- `pCallbackMsgProc` — указатель на функцию обратного вызова. Если функция установлена, она будет вызываться при получении сообщения Windows. Если передано значение `NULL`, DXUT не будет сообщать приложению о поступлении сообщений Windows.
- `pUserContext` — указатель на определяемое пользователем значение, которое передается функции. Значение по умолчанию — `NULL`.

Выполнение стандартных задач

DXUTKillTimer

Удаляет установленный таймер. Прототип функции:

```
HRESULT DXUTKillTimer(
  UINT nIDEvent
);
```

Параметр:

- `nIDEvent` — идентификатор события удаляемого таймера. Это идентификатор, который вернул приложению метод `DXUTSetTimer()`.

DXUTPause

Переводит в режим паузы внутренний таймер DXUT и/или процесс визуализации. Прототип функции:

```
VOID DXUTPause(
  BOOL bPauseTime,
  BOOL bPauseRendering
);
```

Параметры:

- `bPauseTime` — если передано значение `TRUE`, внутренний таймер DXUT переводится в режим паузы. Если передано значение `FALSE`, таймер возвращается из режима паузы.
- `bPauseRendering` — если передано значение `TRUE`, DXUT перестает вызывать функцию обратного вызова `LPDXUTCALLBACKD3D10FRAMERENDER`, но вызов функции `LPDXUTCALLBACKFRAMEMOVE` будет осуществляться дальше. Если передано значение `FALSE`, визуализация продолжится, и режим паузы будет снят.

DXUTResetFrameworkState

Сбрасывает все настройки DXUT в исходное состояние по умолчанию. Все сделанные изменения при этом теряются. Прототип функции:

```
VOID DXUTResetFrameworkState();
```

Параметры: отсутствуют.

Эту функцию обычно вызывать не требуется. Ее используют при тестировании, чтобы вернуть первоначальное состояние DXUT без необходимости перезапуска программы.

DXUTSetConstantFrameTime

Разрешает или запрещает постоянное время вывода кадра. Прототип функции:

```
HRESULT DXUTSetConstantFrameTime(
  BOOL bEnabled,
  FLOAT fTimePerFrame
);
```

Параметры:

- `bEnabled` — флаг, устанавливающий режим постоянного времени кадра. Если передано значение `TRUE`, параметр fTime функции `LPDXUTCALLBACKFRAMEMOVE` и функции прорисовки экрана будет изменяться на фиксированное значение от кадра к кадру.
- `fTimePerFrame` — время на кадр, в секундах. Значение по умолчанию составляет `0.0333f`, таким образом, параметр `fTime` функции `LPDXUTCALLBACKFRAMEMOVE` и функции прорисовки экрана увеличивается на одну секунду через каждые 30 кадров.

Функция имитирует цикл визуализации с фиксированным количеством кадров в секунду путем установки фиксированного значения времени, прошедшего с прорисовки последнего кадра. По умолчанию на одну секунду приходится 30 кадров. Само приложение продолжает прорисовывать кадры на нерегулируемой скорости, которая может быть значительно выше установленной. Эта функция полезна для сохранения выведенных кадров в видеофайле для последующего воспроизведения, так как она позволяет выводить анимацию на указанной скорости, независимо от той скорости, с которой фактически рисуются кадры.

DXUTSetCursorSettings

Устанавливает параметры использования курсора в полноэкранном режиме. Прототип функции:

```
HRESULT DXUTSetCursorSettings(
  bool bShowCursorWhenFullScreen,
  bool bClipCursorWhenFullScreen
);
```

Параметры:

- `bShowCursorWhenFullScreen` — флаг видимости курсора, который указывает, нужно ли отображать курсор в полноэкранном режиме. По умолчанию курсор в полноэкранном режиме не отображается.

- `bClipCursorWhenFullScreen` — флаг ограничения, он указывает, нужно ли ограничивать передвижение курсора границами экрана в полноэкранном режиме.

DXUTSetD3DVersionSupport

Устанавливает флаги, определяющие, может ли приложение использовать Direct3D 9 и/или Direct3D 10. Прототип функции:

```
void DXUTSetD3DVersionSupport(
  bool bAppCanUseD3D9,
  bool bAppCanUseD3D10
);
```

Параметры:

- `bAppCanUseD3D9` — если установлено значение `FALSE`, каркас DXUT не будет вызывать методы Direct3D 9, а также обнаруживать устройства Direct3D 9.
- `bAppCanUseD3D10` — если установлено значение `FALSE`, каркас DXUT не будет вызывать методы Direct3D 10, а также обнаруживать устройства Direct3D 10.

Обычно вызывать эту функцию необходимости нет. DXUT делает вывод, какую версию API поддерживает приложение, исходя из того, какие функции обратного вызова были установлены. Например, если в приложении установлены функции обратного вызова Direct3D 9, DXUT "считает", что приложение поддерживает Direct3D 9. Если необходимо подменить эту логику, приложение может использовать данную функцию, чтобы установить параметры явным образом.

DXUTSetHotkeyHandling

Определяет, активны или нет некоторые "горячие клавиши". Прототип функции:

```
void DXUTSetHotkeyHandling(
  bool bAltEnterToToggleFullscreen,
  bool bEscapeToQuit,
  bool bPauseToToggleTimePause
);
```

Параметры:

- `bAltEnterToToggleFullscreen` — если установлено значение `TRUE`, нажатие комбинации клавиш <Alt>+<Enter> будет переключать приложение

между оконным и полноэкранным режимами. Значение по умолчанию — TRUE.

- bEscapeToQuit — если установлено значение TRUE, программа будет завершаться по нажатию клавиши <Esc>. Значение по умолчанию — TRUE.
- bPauseToToggleTimePause — если установлено значение TRUE, нажатие клавиши <Pause> будет переводить таймер DXUT в режим паузы.

> **Примечание**
>
> Для профессиональных программ, скорее всего, будет предпочтительнее передавать значение TRUE для параметра bAltEnterToToggleFullscreen и значения FALSE для параметров bEscapeToQuit и bPauseToToggleTimePause, в то время как для простых графических программ будет лучше оставить значения TRUE для всех параметров.

DXUTSetMultimonSettings

Устанавливает параметры для работы DXUT при использовании нескольких мониторов. Прототип функции:

```
VOID DXUTSetMultimonSettings(
  BOOL bAutoChangeAdapter
);
```

Параметр:

- bAutoChangeAdapter — если установлено значение TRUE и окно приложения перемещено на другой монитор, DXUT автоматически переключится на использование видеоадаптера другого монитора и будут вызваны функции обратного вызова устройства Direct3D для восстановления изображения сцены.

DXUTSetShortcutKeySettings

С помощью этой функции можно задать, разрешает или запрещает DXUT использовать клавиатурные комбинации Windows и специальные возможности клавиатуры (залипание клавиш, фильтрация ввода, озвучивание переключения режимов). Прототип функции:

```
void DXUTSetShortcutKeySettings(
  bool bAllowWhenFullscreen,
  bool bAllowWhenWindowed
);
```

Параметры:

- `bAllowWhenFullscreen` — если установлено значение `TRUE`, клавиатурные комбинации и специальные возможности в полноэкранном режиме будут разрешены. Значение по умолчанию — `FALSE`.
- `bAllowWhenWindowed` — если установлено значение `TRUE`, клавиатурные комбинации и специальные возможности в оконном режиме будут разрешены. Значение по умолчанию — `TRUE`.

DXUTSetTimer

Запускает таймер DXUT, который через равные интервалы времени будет запускать указанную функцию обратного вызова. Прототип функции:

```
HRESULT DXUTSetTimer(
  LPDXUTCALLBACKTIMER pCallbackTimer,
  FLOAT fTimeoutInSecs,
  UINT *pnIDEvent,
  void *pCallbackUserContext
);
```

Параметры:

- `pCallbackTimer` — указатель на функцию обратного вызова таймера. Функция будет вызываться через период времени, указанный в параметре `fTimeoutInSecs`.
- `fTimeoutInSecs` — интервал в секундах между последовательными вызовами установленной функции обратного вызова таймера. По умолчанию установлено значение `1.0f`.
- `pnIDEvent` — необязательный указатель на переменную, в которую будет помещен идентификатор события, которое генерируется новым таймером. Этот идентификатор будет передаваться функции обратного вызова для указания, от какого именно таймера поступило событие. Таким образом можно использовать одну функцию обратного вызова для нескольких таймеров. Значение по умолчанию — `NULL`.
- `pCallbackUserContext` — указатель на определяемое пользователем значение, которое передается функции. Значение по умолчанию — `NULL`.

DXUTSetWindowSettings

Задает параметры обработки сообщений окна DXUT. Прототип функции:

```
void DXUTSetWindowSettings(
  bool bCallDefWindowProc
);
```

Параметры:

- `bCallDefWindowProc` — если установлено значение `TRUE`, DXUT будет вызывать оконную процедуру по умолчанию, в противном случае оконная процедура вызываться не будет. Значение по умолчанию — `TRUE`.

> ***Примечание***
>
> Обычно эту функцию вызывать не требуется. Она позволяет приложениям, использующим диалоговое окно, обрабатывать сообщения DXUT. Сообщения, отправляемые диалоговому окну, не должны передаваться оконной процедуре по умолчанию. Можно вызвать данную функцию, чтобы изменить такое поведение программы.

DXUTShutdown

Вызывает завершение работы программы и освобождение занятых ресурсов DXUT. Прототип функции:

```
VOID DXUTShutdown(
  int nExitCode
);
```

Параметр:

- `nExitCode` — код завершения программы, возвращаемый с помощью функции `DXUTGetExitCode()`. Значение по умолчанию — 0.

DXUTToggleFullscreen

Переключает приложение из оконного режима в полноэкранный и обратно. Прототип функции:

```
HRESULT DXUTToggleFullscreen();
```

Параметры: отсутствуют.

DXUTToggleREF

Переключает приложение от использования устройства с аппаратной поддержкой (HAL) на устройство с программной эмуляцией графического конвейера (REF) и обратно. Прототип функции:

```
HRESULT DXUTToggleREF();
```

Параметры: отсутствуют.

Получение сведений об установках DXUT

DXUTDoesAppSupportD3D9

Показывает, поддерживает ли приложение вызовы методов Direct3D 9. Прототип функции:

```
bool DXUTDoesAppSupportD3D9();
```

Параметры: отсутствуют.

Функция возвращает значение `TRUE`, если приложение поддерживает вызов методов Direct3D 9, и значение `FALSE` в противном случае.

DXUTDoesAppSupportD3D10

Показывает, поддерживает ли приложение вызовы методов Direct3D 10. Прототип функции:

```
bool DXUTDoesAppSupportD3D10();
```

Параметры: отсутствуют.

Функция возвращает значение `TRUE`, если приложение поддерживает вызов методов Direct3D 10, и значение `FALSE` в противном случае.

DXUTFindValidDeviceSettings

Подбирает допустимые параметры, которые будут использованы при создании нового устройства Direct3D. Прототип функции:

```
HRESULT DXUTFindValidDeviceSettings(
  DXUTDeviceSettings *pOut,
  DXUTDeviceSettings *pIn,
  DXUTMatchOptions *pMatchOptions
);
```

Параметры:

- `pOut` — указатель на структуру типа `DXUTDeviceSettings`, которая будет содержать результат работы функции: допустимые настройки для создания нового устройства.
- `pIn` — указатель на структуру типа `DXUTDeviceSettings`, которая содержит желаемые параметры нового устройства. Значение по умолчанию — `NULL`.
- `pMatchOptions` — указатель на структуру типа `DXUTMatchOptions`, которая содержит флаги, указывающие, как использовать переданные желаемые параметры устройства при подборе параметров нового устройства. Будет

выбран оптимальный вариант, исходя из совпадения параметров, указанных в pMatchOptions. Если передано значение NULL, функция действует, как если бы все поля структуры содержали значения DXUTMT_IGNORE_INPUT, это означает, что функция вернет корректные параметры устройства, максимально приближенные к параметрам устройства по умолчанию. Значение по умолчанию — NULL.

Функция пытается определить допустимый набор параметров, исходя из заданных в параметре pIn настроек. Для каждого параметра имеется критерий сравнения, хранящийся в структуре DXUTMatchOptions. Используя их, функция принимает решения о выборе значений. Функция работает как с параметрами устройств Direct3D 9, так и Direct3D 10.

DXUTGetAutomation

Указывает, был ли использован в командной строке параметр "automation" при запуске программы. Прототип функции:

```
bool DXUTGetAutomation();
```

Параметры: отсутствуют.

DXUTGetD3D10GetDepthStencilView

Получает указатель на интерфейс ID3D10DepthStencilView текущего устройства Direct3D 10. Прототип функции:

```
ID3D10DepthStencilView *DXUTGetD3D10GetDepthStencilView();
```

Параметры: отсутствуют.

DXUTGetD3D10Device

Получает указатель на интерфейс ID3D10Device текущего устройства Direct3D. Прототип функции:

```
ID3D10Device *DXUTGetD3D10Device();
```

Параметры: отсутствуют.

DXUTGetRenderTargetView

Получает указатель на интерфейс ID3D10RenderTargetView текущего устройства Direct3D 10. Прототип функции:

```
ID3D10RenderTargetView *DXUTGetRenderTargetView();
```

Параметры: отсутствуют.

DXUTGetD3D9BackBufferSurfaceDesc

Получает указатель на структуру типа `D3DSURFACE_DESC`, содержащую описание текущего вторичного буфера Direct3D 9. Прототип функции:

```
CONST D3DSURFACE_DESC *DXUTGetD3D9BackBufferSurfaceDesc();
```

Параметры: отсутствуют.

DXUTGetD3D9Device

Получает указатель на интерфейс `IDirect3DDevice9`, представляющий текущее устройство Direct3D. Прототип функции:

```
IDirect3DDevice9 *DXUTGetD3D9Device();
```

Параметры: отсутствуют.

Функция возвращает указатель на интерфейс `IDirect3DDevice9`, представляющий текущее устройство Direct3D, и значение `NULL`, если устройство отсутствует. Счетчик ссылок для данного интерфейса не увеличивается, поэтому вызывающая функция не должна освобождать полученный указатель.

DXUTGetD3D9DeviceCaps

Получает указатель на структуру `D3DCAPS9`, содержащую информацию о возможностях, поддерживаемых видеокартой. Прототип функции:

```
CONST D3DCAPS9 *DXUTGetD3D9DeviceCaps();
```

Параметры: отсутствуют.

Функция возвращает указатель на структуру `D3DCAPS9`, содержащую информацию о возможностях, поддерживаемых видеокартой. Если устройство Direct3D 9 не обнаружено, структура заполняется нулями.

DXUTGetD3DObject

Получает указатель на объект `IDirect3D9`. Прототип функции:

```
IDirect3D9 *DXUTGetD3DObject();
```

Параметры: отсутствуют.

Функция возвращает указатель на объект `IDirect3D9`, если устройство не было создано, возвращается значение `NULL`. Счетчик ссылок для данного интерфейса не увеличивается, поэтому вызывающая функция не должна освобождать полученный указатель.

DXUTGetPresentParameters

Получает параметры показа графики устройства Direct3D 9. Прототип функции:

```
D3DPRESENT_PARAMETERS DXUTGetPresentParameters();
```

Параметры: отсутствуют.

Функция возвращает структуру с параметрами отображения. Если устройство Direct3D 9 не создано, структура заполняется нулями.

DXUTGetDeviceSettings

Получает структуру `DXUTDeviceSettings`, использованную при создании текущего устройства. Прототип функции:

```
DXUTDeviceSettings DXUTGetDeviceSettings();
```

Параметры: отсутствуют.

Структура параметров может описывать как устройство Direct3D 9, так и Direct3D 10, в зависимости от того, что указано в соответствующем поле структуры `DXUTDeviceSettings`. Если устройство не создано, структура заполняется нулями.

DXUTGetDeviceStats

Получает указатель на строку, описывающую текущий используемый тип устройства, режим обработки вершин и название устройства. Прототип функции:

```
LPCWSTR DXUTGetDeviceStats();
```

Параметры: отсутствуют.

DXUTGetDXGIBackBufferSurfaceDesc

Получает указатель на структуру типа `DXGI_SURFACE_DESC`, содержащую описание поверхности вторичного буфера используемого видеоадаптера. Прототип функции:

```
CONST DXGI_SURFACE_DESC *DXUTGetDXGIBackBufferSurfaceDesc();
```

Параметры: отсутствуют.

Если устройство не создано, структура заполняется нулями.

DXUTGetDXGIFactory

Получает указатель на интерфейс `IDXGIFactory`. Прототип функции:

```
IDXGIFactory *DXUTGetDXGIFactory();
```

Параметры: отсутствуют.

Функция предоставляет доступ к глобальному объекту `IDXGIFactory`. Если устройство не создано, возвращается значение `NULL`. Счетчик ссылок для данного интерфейса не увеличивается, поэтому вызывающая функция не должна освобождать полученный указатель.

DXUTGetDXGISwapChain

Получает указатель на интерфейс `IDXGISwapChain` текущего устройства Direct3D 10. Прототип функции:

```
IDXGISwapChain *DXUTGetDXGISwapChain();
```

Параметры: отсутствуют.

Функция предоставляет доступ к глобальному объекту `IDXGISwapChain`. Если устройство не создано, возвращается значение `NULL`. Счетчик ссылок для данного интерфейса не увеличивается, поэтому вызывающая функция не должна освобождать полученный указатель.

DXUTGetElapsedTime

Получает время, прошедшее с вывода последнего кадра, задаваемое в секундах. Прототип функции:

```
FLOAT DXUTGetElapsedTime();
```

Параметры: отсутствуют.

DXUTGetExitCode

Получает код завершения приложения DXUT. Прототип функции:

```
INT DXUTGetExitCode();
```

Параметры: отсутствуют.

Функция возвращает одно из значений, приведенных в табл. П2.2.

Таблица П2.2. *Возможные значения кода завершения приложения DXUT*

Код	Описание
0	Успешное выполнение
1	Возникла неопознанная ошибка
2	Не обнаружено устройство Direct3D с указанными параметрами
3	Не найден файл с данными

Таблица П2.2 *(окончание)*

Код	Описание
4	Невозможно открыть файл с данными
5	При попытке создания устройства Direct3D возникла ошибка
6	При попытке сброса устройства Direct3D возникла ошибка
7	При выполнении функции обратного вызова создания устройства возникла ошибка
8	При выполнении функции обратного вызова сброса устройства возникла ошибка
9	Установлена неправильная версия Direct3D или D3DX
10	Последнее используемое устройство до выхода было устройство REF
11	Устройство было удалено

Примечание

Код завершения, возвращаемый этой функцией, обычно используется в качестве кода завершения приложения в функции `WinMain()`.

DXUTGetFPS

Получает количество выводимых кадров в секунду. Прототип функции:

```
FLOAT DXUTGetFPS();
```

Параметры: отсутствуют.

DXUTGetFrameStats

Получает указатель на строку, содержащую количество выводимых кадров в секунду (необязательно), разрешение, формат вторичного буфера и формат шаблонного буфера глубины. Прототип функции:

```
LPCWSTR DXUTGetFrameStats(
  bool bIncludeFPS
);
```

Параметр:

- `bIncludeFPS` — если установлено значение `TRUE`, возвращаемая строка будет содержать количество кадров в секунду. В противном случае количество кадров выводиться не будет.

DXUTGetFullsceenClientRectAtModeChange

Функция возвращает клиентскую область окна приложения при полноэкранном режиме в виде структуры `RECT`. Прототип функции:

```
CONST RECT *DXUTGetFullsceenClientRectAtModeChange();
```

Параметры: отсутствуют.

Если приложение находится в полноэкранном режиме, возвращается клиентская область окна приложения. Если в оконном — возвращается клиентская область окна, сохраненная во время последнего включения полноэкранного режима.

DXUTGetHINSTANCE

Получает описатель экземпляра приложения. Прототип функции:

```
HINSTANCE DXUTGetHINSTANCE();
```

Параметры: отсутствуют.

DXUTGetHWND

Получает описатель окна текущего устройства. Прототип функции:

```
HWND DXUTGetHWND();
```

Параметры: отсутствуют.

DXUTGetHWNDDeviceFullScreen

Получает описатель окна устройства, которое используется при работе в полноэкранном режиме. Прототип функции:

```
HWND DXUTGetHWNDDeviceFullScreen();
```

Параметры: отсутствуют.

DXUTGetHWNDDeviceWindowed

Получает описатель окна устройства, которое используется при работе в оконном режиме. Прототип функции:

```
HWND DXUTGetHWNDDeviceWindowed();
```

Параметры: отсутствуют.

DXUTGetHWNDFocus

Получает описатель окна, на которое установлен фокус. Прототип функции:

```
HWND DXUTGetHWNDFocus();
```

Параметры: отсутствуют.

Окно, на которое установлен фокус ввода, может дать информацию о том, что окно приложения используется в фоновом режиме, т. е. произошло переключение при помощи комбинации <Alt>+<Tab>, щелчка мыши или каким-то другим способом. Окно, на которое установлен фокус, обычно то же самое, что и окно устройства.

DXUTGetShowMsgBoxOnError

Сообщает, будет ли DXUT выводить сообщения об ошибках. Прототип функции:

```
BOOL DXUTGetShowMsgBoxOnError();
```

Параметры: отсутствуют.

Если DXUT будет выводить сообщения при возникновении ошибок, возвращается значение `TRUE`.

DXUTGetTime

Получает текущее время в секундах. Прототип функции:

```
DOUBLE DXUTGetTime();
```

Параметры: отсутствуют.

DXUTGetWindowClientRect

Получает размеры текущей клиентской области окна приложения в виде структуры `RECT`. Прототип функции:

```
CONST RECT *DXUTGetWindowClientRect();
```

Параметры: отсутствуют.

DXUTGetWindowClientRectAtModeChange

Получает клиентскую область окна приложения при оконном режиме в виде структуры `RECT`. Прототип функции:

```
CONST RECT *DXUTGetWindowClientRectAtModeChange();
```

Параметры: отсутствуют.

Если приложение находится в оконном режиме, возвращается клиентская область окна приложения. Если в полноэкранном — возвращается клиентская область окна, сохраненная во время последнего включения оконного режима.

DXUTGetWindowTitle

Получает указатель на строку, содержащую текст заголовка окна приложения. Прототип функции:

```
LPCWSTR DXUTGetWindowTitle();
```

Параметры: отсутствуют.

DXUTIsActive

Сообщает об активности приложения. Прототип функции:

```
BOOL DXUTIsActive();
```

Параметры: отсутствуют.

DXUT не активен, когда окно приложения теряет фокус. Если DXUT не активен, он уступает процессорное время другим приложениям. Если DXUT не активен в полноэкранном режиме, визуализация прекращается на время, пока окно приложения находится в свернутом состоянии.

DXUTIsAppRenderingWithD3D9

Сообщает, выполняется ли визуализация в приложении устройством Direct3D 9. Прототип функции:

```
BOOL DXUTIsAppRenderingWithD3D9();
```

Параметры: отсутствуют.

Функция возвращает значение `TRUE`, если визуализация выполняется устройством Diret3D 9, и `FALSE` в противном случае.

DXUTIsAppRenderingWithD3D10

Сообщает, выполняется ли визуализация в приложении устройством Direct3D 10. Прототип функции:

```
BOOL DXUTIsAppRenderingWithD3D10();
```

Параметры: отсутствуют.

Функция возвращает значение `TRUE`, если визуализация выполняется устройством Direct3D 10, и `FALSE` в противном случае.

DXUTIsD3D10Available

Сообщает, имеется ли в системе возможность использования API Direct3D 10. Прототип функции:

```
BOOL DXUTIsD3D10Available();
```

Параметры: отсутствуют.

Функция возвращает значение TRUE, если возможно использование API Direct3D 10, однако это не гарантирует наличие в системе видеокарты, совместимой с Direct3D 10.

DXUTIsKeyDown

Сообщает, нажата или отпущена указанная клавиша в момент вызова функции. Прототип функции:

```
BOOL DXUTIsKeyDown(
  BYTE vKey
);
```

Параметры:

- vKey — виртуальный код клавиши клавиатуры. В качестве примеров кодов клавиш можно привести: VK_F1 (клавиша <F1>), VK_LSHIFT (левая клавиша <Shift>), VK_RCONTROL (правая клавиша <Ctrl>), VK_RMENU (правая клавиша <Menu>) и 41 (представляет клавишу <A>).

Функция возвращает значение TRUE, если указанная клавиша нажата, и FALSE в противоположном случае.

DXUTIsMouseButtonDown

Сообщает, нажата или отпущена указанная кнопка мыши в момент вызова функции. Прототип функции:

```
BOOL DXUTIsMouseButtonDown(
  BYTE vButton
);
```

Параметры:

- vKey — виртуальный код кнопки мыши. Допустимые значения: VK_LBUTTON (левая кнопка), VK_RBUTTON (правая кнопка), VK_MBUTTON (средняя кнопка), VK_XBUTTON1 (первая дополнительная кнопка) и VK_XBUTTON2 (вторая дополнительная кнопка).

Функция возвращает значение TRUE, если указанная кнопка нажата, и FALSE в противном случае.

DXUTIsRenderingPaused

Сообщает, находится ли визуализация в DXUT в режиме паузы. Прототип функции:

```
BOOL DXUTIsRenderingPaused();
```

Параметры: отсутствуют.

Функция возвращает значение `TRUE`, если визуализация находится в режиме паузы.

DXUTIsTimePaused

Сообщает, находится ли таймер DXUT в режиме паузы. Прототип функции:

```
BOOL DXUTIsTimePaused();
```

Параметры: отсутствуют.

Функция возвращает значение `TRUE`, если таймер находится в режиме паузы.

DXUTIsWindowed

Сообщает, находится ли приложение в оконном режиме. Прототип функции:

```
BOOL DXUTIsWindowed();
```

Параметры: отсутствуют.

Функция возвращает значение `TRUE`, если приложение находится в оконном режиме. Если приложение не находится в оконном режиме либо отсутствует устройство, возвращается значение `FALSE`.

Структуры

DXUTD3D9DeviceSettings

Описывает настройки, используемые при создании устройства Direct3D 9. Определение структуры:

```
typedef struct DXUTD3D9DeviceSettings {
    UINT AdapterOrdinal;
    D3DDEVTYPE DeviceType;
    D3DFORMAT AdapterFormat;
    DWORD BehaviorFlags;
    D3DPRESENT_PARAMETERS pp;
} DXUTD3D9DeviceSettings, *LPDXUTD3D9DeviceSettings;
```

Значение полей:

- `AdapterOrdinal` — порядковый номер видеоадаптера устройства Direct3D 9;
- `DeviceType` — тип устройства;
- `AdapterFormat` — формат поверхности видеоадаптера;

- `BehaviorFlags` — флаги, определяющие функционирование устройства Direct3D 9, можно использовать комбинацию из нескольких значений типа `D3DCREATE`;
- `pp` — структура, описывающая параметры представления изображения.

DXUTD3D10DeviceSettings

Описывает настройки, относящиеся к созданию и использованию устройства Direct3D 10. Определение структуры:

```
typedef struct DXUTD3D10DeviceSettings {
    UINT AdapterOrdinal;
    D3D10_DRIVER_TYPE DriverType;
    UINT Output;
    DXGI_SWAP_CHAIN_DESC sd;
    UINT32 CreateFlags;
    UINT32 SyncInterval;
    DWORD PresentFlags;
    BOOL AutoCreateDepthStencil;
    DXGI_FORMAT AutoDepthStencilFormat;
} DXUTD3D10DeviceSettings, *LPDXUTD3D10DeviceSettings;
```

Значение полей:

- `AdapterOrdinal` — порядковый номер видеоадаптера устройства Direct3D 10;
- `DriverType` — тип драйвера устройства Direct3D 10;
- `Output` — порядковый номер порта вывода видеоадаптера;
- `sd` — структура, описывающая цепочку переключений;
- `CreateFlags` — флаги, используемые при создании устройства Direct3D 10;
- `SyncInterval` — поле для внутреннего использования, применяется для синхронизации с кадровой разверткой монитора;
- `PresentFlags` — флаги представления изображения на экране;
- `AutoCreateDepthStencil` — флаг, определяющий необходимость создания ресурса и представления данных как шаблонного буфера глубины, которые будут созданы DXUT автоматически, если установлено значение `TRUE`;
- `AutoDepthStencilFormat` — формат шаблонного буфера глубины.

> **Примечание**
>
> Структуры, используемые выше в качестве полей, описаны в *Приложении 1*.

DXUTDeviceSettings

Объединение, описывающее свойства создаваемого устройства, которое определяется следующим образом:

```
typedef struct DXUTDeviceSettings {
    DXUTDeviceVersion ver;
    union {
        DXUTD3D9DeviceSettings d3d9;
        DXUTD3D10DeviceSettings d3d10;
    };
} DXUTDeviceSettings, *LPDXUTDeviceSettings;
```

Значение полей:

- `ver` — поле, которое указывает, для какой версии API создается устройство: для Direct3D 9 или Direct3D 10;
- `d3d9` — настройки устройства Direct3D 9, действительны только в том случае, если параметр `ver` равен `DXUT_D3D9_DEVICE`;
- `d3d10` — настройки устройства Direct3D 10, действительны только в том случае, если параметр `ver` равен `DXUT_D3D10_DEVICE`.

DXUTMatchOptions

Описывает метод сравнения при поиске допустимых параметров устройства Direct3D с помощью функции `DXUTFindValidDeviceSettings()`. Каждое поле данной структуры соответствует полю в структуре `DXUTDeviceSettings`. Если поле содержит значение `DXUTMT_IGNORE_INPUT`, в соответствующем поле структуры `DXUTDeviceSettings` подставляется значение по умолчанию. Определение структуры `DXUTMatchOptions`:

```
typedef struct DXUTMatchOptions {
    DXUT_MATCH_TYPE eAPIVersion;
    DXUT_MATCH_TYPE eAdapterOrdinal;
    DXUT_MATCH_TYPE eOutput;
    DXUT_MATCH_TYPE eDeviceType;
    DXUT_MATCH_TYPE eWindowed;
    DXUT_MATCH_TYPE eAdapterFormat;
    DXUT_MATCH_TYPE eVertexProcessing;
    DXUT_MATCH_TYPE eResolution;
    DXUT_MATCH_TYPE eBackBufferFormat;
    DXUT_MATCH_TYPE eBackBufferCount;
    DXUT_MATCH_TYPE eMultiSample;
```

```
    DXUT_MATCH_TYPE eSwapEffect;
    DXUT_MATCH_TYPE eDepthFormat;
    DXUT_MATCH_TYPE eStencilFormat;
    DXUT_MATCH_TYPE ePresentFlags;
    DXUT_MATCH_TYPE eRefreshRate;
    DXUT_MATCH_TYPE ePresentInterval;
} DXUTMatchOptions, *LPDXUTMatchOptions;
```

Значения полей представляют собой методы сравнения следующих полей:

- `eAPIVersion` — номеров версий API;
- `eAdapterOrdinal` — порядковых номеров видеоадаптера устройства Direct3D;
- `eOutput` — порядковых номеров портов вывода видеоадаптера;
- `eDeviceType` — типов устройств, значение `DXUTMT_IGNORE_INPUT` означает использование типа `D3DDEVTYPE_HAL`;
- `eWindowed` — параметров оконного либо полноэкранного режимов, при указании значения `DXUTMT_IGNORE_INPUT` выбирается оконный режим (`TRUE`);
- `eAdapterFormat` — установленных форматов видеоадаптеров; значение `DXUTMT_IGNORE_INPUT` предписывает использование формата, установленного для фона рабочего стола, либо формата `D3DFMT_X8R8G8B8`, если глубина цвета экрана меньше 32-х бит;
- `eVertexProcessing` — флагов обработки вершин, т. е `D3DCREATE_HARDWARE_VERTEXPROCESSING` (аппаратная обработка), `D3DCREATE_MIXED_VERTEXPROCESSING` (смешанная обработка) или `D3DCREATE_SOFTWARE_VERTEXPROCESSING` (программная обработка);
- `eResolution` — разрешений устройства, константа `DXUTMT_IGNORE_INPUT` означает разрешение 640×480 для оконного режима или установленного разрешения экрана для полноэкранного режима;
- `eBackBufferFormat` — форматов вторичных буферов, значение `DXUTMT_IGNORE_INPUT` приводит к использованию параметра, указанного в формате видеоадаптера;
- `eBackBufferCount` — количества вторичных буферов, указание `DXUTMT_IGNORE_INPUT` приводит к использованию значения 2 для тройной буферизации;
- `eMultiSample` — параметров сглаживания изображения при помощи мультисэмплинга, значение `DXUTMT_IGNORE_INPUT` означает выключение мультисэмплинга (`MultiSampleQuality = 0`);

- `eSwapEffect` — способов взаимодействия буферов при их переключении, передача значения `DXUTMT_IGNORE_INPUT` приводит к использованию флага `D3DSWAPEFFECT_DISCARD`;
- `eDepthFormat` — форматов буферов глубины, которые устройство создает автоматически, если установлено значение `DXUTMT_IGNORE_INPUT`, то по умолчанию используется либо `D3DFMT_D16`, если глубина цвета вторичного буфера 16 бит или меньше, либо `D3DFMT_D32` в противном случае;
- `eStencilFormat` — форматов буферов шаблона, которые устройство создает автоматически, константа `DXUTMT_IGNORE_INPUT` приводит к использованию формата `D3DFMT_D16`, если глубина цвета вторичного буфера не превосходит 16 бит, или `D3DFMT_D32` в противном случае;
- `ePresentFlags` — флагов представления изображения на экране, если установлено значение `DXUTMT_IGNORE_INPUT`, флагом по умолчанию будет `D3DPRESENTFLAG_DISCARD_DEPTHSTENCIL`;
- `eRefreshRate` — частот обновления экрана, константа `DXUTMT_IGNORE_INPUT` означает использование оконного режима;
- `ePresentInterval` — промежутков между сменами кадров, передача `DXUTMT_IGNORE_INPUT` означает установку значений `D3DPRESENT_INTERVAL_IMMEDIATE` и `D3DPRESENT_INTERVAL_DEFAULT` для оконного и полноэкранного режима, соответственно.

Возможные значения полей структуры представлены в табл. П2.3.

Таблица П2.3. *Возможные значения полей структуры* `DXUTMatchOptions`

Значение	Описание
`DXUTMT_IGNORE_INPUT`	Игнорировать переданное желаемое значение и установить значение параметра устройства как можно ближе к значению по умолчанию
`DXUTMT_PRESERVE_INPUT`	Вернуть переданное желаемое значение без изменений
`DXUTMT_CLOSEST_TO_INPUT`	Вернуть значение, как можно более близкое к желаемому значению, переданному на вход функции

Приложение 3

Описание компакт-диска

Папки	Описание
\Glava2	Исходный текст примеров *главы 2*: минимальное приложение Windows и минимальное приложение Direct3D 10
\Glava4	Исходный текст примеров *главы 3*: вывод на экран текста и спрайта
\Glava5	Исходный текст примеров *главы 5*: первый файл эффектов
\Glava6	Исходный текст примеров *главы 6*: вывод на экран треугольника, анимация треугольника, вывод треугольника в цвете, работа в полноэкранном режиме
\Glava7	Исходный текст примеров *главы 7*: работа с источниками света и наложение текстуры на объект
\Glava8	Исходный текст примеров *главы 8*: геометрический шейдер
\Glava9	Исходный текст примеров *главы 9*: вывод текста при помощи DXUT
\Glava10	Исходный текст примеров *главы 10*: загрузка трехмерной модели и использование камеры DXUT для просмотра модели
\Glava11	Исходный текст примеров *главы 11*: работа с элементами интерфейса DXUT
\Glava12	Исходный текст примеров *главы 12*: создание диалоговых окон и использование окна настроек параметров устройства Direct3D

Предметный указатель

A, C

API 9
CALLBACK 36
COM 41

D

Direct3D 7
DirectDraw 7
DirectInput 7
DirectMusic 8
DirectPlay 8
DirectShow 8
DirectSound 7
DirectX 7
DirectX Sample Browser 279
DXGI 64
DXUT 279

G, H

GUID 52
HAL 9
HLSL:
  базовые типы переменных 143
  векторы 144
  встроенные функции 151
  матрицы 145
  методы потоковых объектов 262
  объявление функций 150
  просмотр сообщений об ошибках 162
  семантика 149
  структуры 150
HRESULT 27

I, V, W

IME (Input Method Editor) 329
Visual Studio 14
WINAPI 25

Б

База знаний Microsoft 8
Буферный ресурс 113

В

Вектор 77
Векторное произведение векторов 82
Вершинный буфер 158
Вершинный шейдер 141
Вторичный буфер 45
Вывод текста 97
Вычисление вектора нормали
  к поверхности 264
Вычитание векторов 79
Вычитание матриц 85

Г

Геометрические преобразования:
  вращение 75
  масштабирование 76
  перемещение 74
Геометрический шейдер 141
Главная диагональ матрицы 87

Д

Двойная буферизация 45
Диффузная модель освещения 210
Добавление файла в проект 17

Е, З

Единичная матрица 87
Закон косинусов Ламберта 210

И

Идентификатор события DXUT 340
Индексный буфер 213
Инициализация Direct3D 10 45
Интерфейс:
ID3D10Blob 381
ID3D10Buffer 382
ID3D10DepthStencilView 383
ID3D10Device 384
ID3D10Effect 389
ID3D10EffectMatrixVariable 390
ID3D10EffectScalarVariable 390
ID3D10EffectTechnique 392
ID3D10EffectVectorVariable 392
ID3D10InputLayout 393
ID3D10RenderTargetView 393
ID3D10ShaderResourceView 394
ID3DX10Font 394
ID3DX10Sprite 122, 396
IDXGIFactory 440
IDXGISwapChain 399, 441
Интерфейс пользователя:
выпадающий список (combobox) 329
кнопка (button) 328
переключатели (radiobutton) 330
поле ввода (editbox) 328
ползунок (slider) 330
список (listbox) 329
статичный текст (static text) 327
флажок (checkbox) 328
Источник света:
направленный 206
прожектор 207
точечный 206

К

Квант времени 21
Класс:
CDXUTDialog 331
CDXUTDialogResourceManager 331
CDXUTSDKMesh 305
CDXUTTextHelper 291
CModelViewerCamera 321
Класс окна 27
Комбинирование преобразований 90
Компиляция 20
Координаты вектора 77

М

Макрос:
__uuidof 52
D3DXToDegree 75
D3DXToRadian 75
FAILED 25
SAFE_DELETE 292
SAFE_RELEASE 292
SUCCEEDED 26
V 291
V_RETURN 290
Материалы 247
Матрица 84
вида 91
вращения 89
масштабирования 90
перемещения 88
проекции 91
Метод:
CD3DSettingsDlg:
IsActive 375
OnD3D10CreateDevice 375
OnD3D10DestroyDevice 377
OnD3D10ResizedSwapChain 377
CDXUTComboBox:
AddItem 356
SetDropHeight 356
CDXUTDialog:
AddButton 339
AddCheckBox 344
AddComboBox 356
AddEditBox 349
AddListBox 359
AddRadioButton 365
AddSlider 362
AddStatic 333
EnableCaption 370
GetButton 342
GetCheckBox 346
GetControl 337
GetHeight 372
GetListBox 359
GetWidth 372
OnRender 334
SetBackgroundColors 370
SetCaptionHeight 370
SetCaptionText 370
SetLocation 370
SetSize 370
SetVisible 370

CDXUTDialogResourceManager:
OnD3D10DestroyDevice 337
CDXUTListBox:
GetItem 360
GetSelectedIndex 360
CDXUTSDKMesh:
Create 305
Destroy 312
Render 310
CDXUTTextHelper:
Begin 293
DrawFormattedText 295
End 294
CModelViewerCamera:
GetWorldMatrix 323
HandleMessages 324
SetButtonMasks 322
SetProjParams 321
SetViewParams 321
SetWindow 322
ID3D10Blob:
GetBufferPointer 163, 381
GetBufferSize 382
ID3D10Buffer:
GetDesc 382
Map 382
Unmap 383
ID3D10DepthStencilView:
GetDesc 383
ID3D10Device:
ClearDepthStencilView 271, 384
ClearRenderTargetView 55, 384
ClearState 56, 385
CreateBuffer 166, 385
CreateDepthStencilView 385
CreateInputLayout 164, 386
CreateRenderTargetView 52, 386
CreateTexture2D 269
Draw 173, 387
DrawIndexed 226, 387
IASetIndexBuffer 219, 388
IASetInputLayout 166, 389
IASetPrimitiveTopology 169, 389
IASetVertexBuffers 168
OMSetRenderTargets 52, 388
RSSetViewports 54
ID3D10Effect:
GetTechniqueByName 163, 389
GetVariableByName 389
ID3D10EffectMatrixVariable:
GetMatrix 390
SetMatrix 390
ID3D10EffectPass:
Apply 173
ID3D10EffectScalarVariable:
SetBool 390
SetFloat 391
SetFloatArray 391
SetInt 391
SetIntArray 391
ID3D10EffectTechnique:
GetDesc 173
GetPassByIndex 392
GetPassByName 392
ID3D10EffectVariable:
AsMatrix 186
ID3D10EffectVectorVariable:
SetFloatVector 393
SetFloatVectorArray 393
ID3D10RenderTargetView:
GetDesc 394
ID3D10ShaderResourceView:
GetDesc 394
ID3DX10Font:
DrawText 102, 394
ID3DX10Sprite:
Begin 122, 396
DrawSpritesBuffered 122, 396
DrawSpritesImmediate 122, 397
End 123, 397
Flush 123, 398
GetDevice 123, 398
GetProjectionTransform 123, 398
GetProjectionTransform 398
GetViewTransform 123
SetProjectionTransform 123, 399
SetViewTransform 123, 399
IDXGIFactory:
MakeWindowAssociation 65
IDXGISwapChain:
GetBuffer 51, 399
GetFullscreenState 66
Present 55, 400
ResizeBuffers 68, 400
SetFullscreenState 66, 401
IUnknown:
Release 56
Мипмаппинг 99
Мировая матрица 91
Модуль вектора 77
Модуль экспорта от Panda Software 300

Н, О

Нормализация вектора 81
Область отображения 46
Общее освещение 205
Окно приложения Windows 27
Операционная система 21
Описатель 25
Очередь сообщений 22

П

Параметры командной строки
  приложения DXUT 425
Первичный буфер 45
Пиксельный шейдер 141
Подресурс 114
Полноэкранный режим 64
Потоковый объект 260
Представление данных 45
Примитив 74
Программа meshconvert 302

Р

Режимы адресации текстур:
  "обертывание" 249
  "отражение" 250
  "рамка" 251
  "фиксирование" 251
Режимы работы:
  растеризатора 194
  сэмплера 256
Ресурсы 21
Ресурсы в Direct3D 113

С

Система координат:
  левосторонняя 72
  правосторонняя 73
Скалярное произведение векторов 81
Сложение векторов 78
Сложение матриц 85
Событие:
  WM_KEYDOWN 66
События 22
Создание нового проекта 15
Сообщение:
  WM_DESTROY 36
  WM_QUIT 37
  WM_SIZE 67
Сообщения Windows 22
Способ построения примитивов 159
Стадии графического конвейера 140
Структура:
  D3D10_BUFFER_DESC 167, 219, 411
  D3D10_DEPTH_STENCIL_VIEW_
    DESC 270, 412
  D3D10_INPUT_ELEMENT_
    DESC 163, 413
  D3D10_PASS_DESC 165, 414
  D3D10_SUBRESOURCE_
    DATA 168, 415
  D3D10_TECHNIQUE_DESC 173
  D3D10_TEXTURE2D_DESC 268, 415
  D3D10_VIEWPORT 53, 416
  D3DX10_IMAGE_INFO 117, 417
  D3DX10_IMAGE_LOAD_
    INFO 116, 418
  D3DX10_SPRITE 125, 419
  D3DXCOLOR 104
  D3DXMATRIX 84
  D3DXVECTOR2 73, 420
  D3DXVECTOR3 73, 420
  D3DXVECTOR4 420
  DXGI_SWAP_CHAIN_DESC 48, 421
  DXUTD3D10DeviceSettings 448
  DXUTD3D9DeviceSettings 447
  DXUTDeviceSettings 449
  DXUTMatchOptions 449
  MSG 35
  RECT 48, 422
  WNDCLASSEX 28
Структура приложения Direct3D 10 42
Структура приложения Windows 23
Структура файла эффектов 143
Структура шейдера 142

Т

Текстура 247
Текстурные координаты 248
Текстурный ресурс 113
Техника отображения 143

У

Умножение вектора на число 80
Умножение матрицы на матрицу 86
Умножение матрицы на число 85
Установка DirectX SDK 11

Ф

Файлы библиотек Direct3D 10 20
Фильтрация текстур 252
Формат .sdkmesh 299
Формат .x 299
Формат вершины 158
Форматирование строки 296
Функция:
CDXUTTextHelper:
DrawTextLine 293
CreateWindow 31
D3D10CreateBlob 401
D3D10CreateDeviceAndSwap
Chain 50, 402
D3DX10CreateEffectFromFile 161, 403
D3DX10CreateFont 98, 404
D3DX10CreateShaderResource-View
FromFile 115, 406
D3DX10CreateSprite 119, 406
D3DX10GetImageInfoFromFile 118, 407
D3DXMatrixIdentity 87, 407
D3DXMatrixLookAtLH 94, 408
D3DXMatrixLookAtRH 94
D3DXMatrixMultiply 87
D3DXMatrixOrthoLH 408
D3DXMatrixPerspectiveFovLH 95, 409
D3DXMatrixRotationX 89, 409
D3DXMatrixRotationY 89, 410
D3DXMatrixRotationZ 89, 410
D3DXMatrixScaling 90, 410
D3DXMatrixTranslation 88, 411
DefWindowProc 36
DispatchMessage 35
DXUTCreateDevice 287, 423
DXUTCreateWindow 286, 424
DXUTDoesAppSupportD3D10 437
DXUTDoesAppSupportD3D9 437
DXUTFindValidDeviceSettings 437
DXUTGetAutomation 438
DXUTGetD3D10DepthStencilView 293
DXUTGetD3D10Device 438
DXUTGetD3D10GetDepthStencilView 438
DXUTGetD3D10RenderTargetView 293
DXUTGetD3D9BackBufferSurfaceDesc 439
DXUTGetD3D9Device 439
DXUTGetD3D9DeviceCaps 439
DXUTGetD3DObject 439
DXUTGetDeviceSettings 309, 440
DXUTGetDeviceStats 295, 440
DXUTGetDXGIBackBufferSurfaceDesc 440
DXUTGetDXGIFactory 440
DXUTGetDXGISwapChain 441
DXUTGetElapsedTime 441
DXUTGetExitCode 441
DXUTGetFPS 295, 442
DXUTGetFrameStats 295, 442
DXUTGetFullsceenClientRectAt
Mode-Change 443
DXUTGetHINSTANCE 443
DXUTGetHWND 443
DXUTGetHWNDDeviceFullScreen 443
DXUTGetHWNDDeviceWindowed 443
DXUTGetHWNDFocus 443
DXUTGetPresentParameters 440
DXUTGetRenderTargetView 438
DXUTGetShowMsgBoxOnError 444
DXUTGetTime 444
DXUTGetWindowClientRect 444
DXUTGetWindowClientRectAt
Mode-Change 444
DXUTGetWindowTitle 445
DXUTInit 285, 424
DXUTIsActive 445
DXUTIsAppRenderingWithD3D10 445
DXUTIsAppRenderingWithD3D9 445
DXUTIsD3D10Available 445
DXUTIsKeyDown 446
DXUTIsMouseButtonDown 446
DXUTIsRenderingPaused 446
DXUTIsTimePaused 447
DXUTIsWindowed 342, 447
DXUTKillTimer 431
DXUTMainLoop 426
DXUTPause 431
DXUTRender3DEnvironment 427
DXUTResetFrameworkState 431
DXUTSetCallbackD3D10Device-
Acceptable 427
DXUTSetCallbackD3D10DeviceCreated 427
DXUTSetCallbackD3D10
Device-Destroyed 428
DXUTSetCallbackD3D10FrameRender 428
DXUTSetCallbackD3D10SwapChain-
Releasing 428
DXUTSetCallbackD3D10Swap
Chain-Resized 429
(окончание рубрики см. на стр. 458)

Функция *(окончание)*:
DXUTSetCallbackDeviceChanging 429
DXUTSetCallbackFrameMove 430
DXUTSetCallbackMsgProc 430
DXUTSetConstantFrameTime 432
DXUTSetCursorSettings 432
DXUTSetCursorSettings 286
DXUTSetD3DVersionSupport 433
DXUTSetHotkeyHandling 433
DXUTSetMultimonSettings 434
DXUTSetShortcutKeySettings 434
DXUTSetTimer 435
DXUTSetWindowSettings 435
DXUTShutdown 436
DXUTShutdown 371
DXUTToggleFullscreen 436
DXUTToggleFullScreen 342
DXUTToggleREF 436
GetClientRect 48
GetMessage 34
GetTickCount 191
InitDirect3D10 46
InitWindow 25
IsD3D10DeviceAcceptable 284
LoadCursor 30
LoadIcon 29
ModifyDeviceSettings 284
MsgProc 284
OnD3D10CreateDevice 285
OnD3D10DestroyDevice 285
OnD3D10FrameRender 285
OnD3D10ReleasingSwapChain 285
OnD3D10ResizedSwapChain 285
OnDeviceRemoved 284
OnFrameMove 284
OnGUIEvent 331
OnKeyboard 284
OnMouse 284
PeekMessage 44
PostQuitMessage 37
RegisterClassEx 30
RenderScene 54
ResizeBuffers 67
ShowWindow 33
StringCchPrintf 351
TranslateMessage 34
UpdateWindow 33
WinMain 24
WndProc 35
wWinMain 282

Ц

Цвет:
зеркальных бликов 247
испускаемого света 247
подсветки 247
рассеянного света 247
Цикл обработки сообщений 33

Ш

Шаблонный буфер глубины 267
Шейдер 139
Шрифтовой объект 98

Э

Экспорт в формат .x 300

Я

Язык HLSL 139