Book Download From WarezTeam.WS

Том Миллер

Managed DirectX*9

Программирование графики и игр

**омпэ*>

Предисловие Боба Гейнса
Менеджера проекта DirectX SDK корпорации Microsoft

SAMS [Pi]

KICK START

È

Managed DirectX 9

KICK START

Управляемый DirectX9 расширяет возможности программирования игр и графики.

Книга позволяет изучить возможности использования Управляемого DirectX при разработке различных графических и мультимедийных приложений. В данной книге рассмотрены как основы программирования 3D-графики, так и более сложные разделы, например, управление уровнями детализации mesh-объектов, использование высокоуровневого языка шейдеров и символьной анимации.

В книге уделено достаточное внимание таким важным вопросам, как управление звуком и устройствами ввода или добавление звуковых эффектов и джойстика с обратной связью.

Материал книги "Managed DirectX 9" направлен в большей степени на непосредственное создание полнофункциональных мультимедийных приложений, нежели на изучение излишнего языкового синтаксиса.

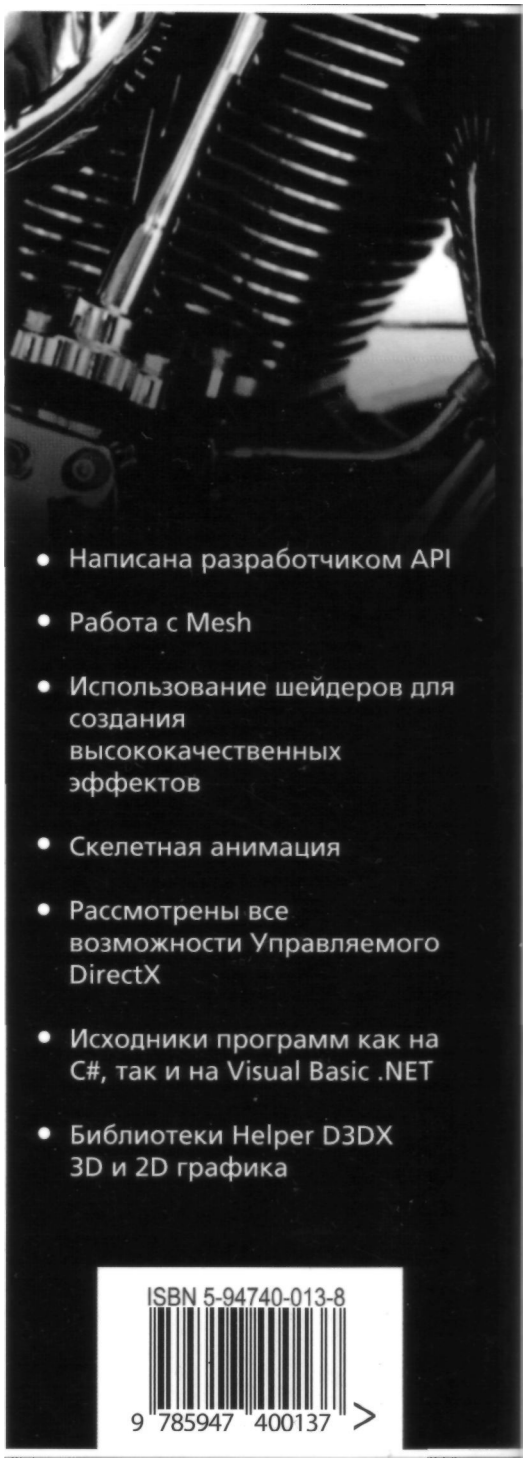
Мы полагаем, что читатель этой книги уже знаком с языком C# (или Visual Basic .NET) и средой .NET Runtime. Автор этой книги имеет огромный опыт разработки приложений с помощью DirectX с Управляемым кодом и является своего рода первоисточником информации в данной области.

ТОМ Миллер является ведущим разработчиком API Managed DirectX. В течении последних четырех лет он участвовал в разработке DirectX API, включая SDK и DirectX для Visual Basic. Перед этим он работал в группе разработчиков Visual Basic и Office корпорации Microsoft.

Содержание прилагаемого CD диска:

- DirectX 9 SDK
- Исходные тексты программ на C# и Visual Basic.NET
- .NET Runtime версии 1.1.

КАТЕГОРИЯ: Программирование.
ОХВАТЫВАЕТ: Программирование графики и компьютерных игр с помощью Управляемого DirectX 9.
УРОВЕНЬ пользователя: Средний.



Том Миллер

DirectX 9 с управляемым кодом

Программирование игр и графика

KICK START

Издательский Дом «КомБук»
Москва, 2005

Содержание

Предисловие....14
Об авторе....15
Посвящается....16
Будем признательны за ваши отзывы и пожелания!....17
Введение....18

ЧАСТЬ I
ВВЕДЕНИЕ В КОМПЬЮТЕРНУЮ ГРАФИКУ

Глава 1. Введение в Direct3D....24
Глава 2. Выбор подходящего устройства....48
Глава 3. Введение в технологию рендеринга....57
Глава 4. Более совершенные технологии рендеринга....76
Глава 5. Рендеринг Mesh-объектов....91
Глава 6. Использование Управляемого DirectX для программирования игр....106

ЧАСТЬ II
ОСНОВНЫЕ КОНЦЕПЦИИ ПОСТРОЕНИЯ ГРАФИКИ

Глава 7. Использование дополнительных свойств и возможностей Mesh-объектов....144
Глава 8. Введение в ресурсы....162

Глава 9. Применение других типов Mesh.....177
Глава 10. Использование вспомогательных классов.....193

ЧАСТЬ III
БОЛЕЕ СОВЕРШЕННЫЕ МЕТОДЫ ПОСТРОЕНИЯ ГРАФИКИ

Глава 11. Введение в программируемый конвейер, язык шейдеров.....216
Глава 12. Использование языка шейдеров HLSL.....'237
Глава 13. Рендеринг скелетной анимации.....257

ЧАСТЬ IV
ЗВУК И УСТРОЙСТВА ВВОДА

Глава 14. Добавление звука.....276
Глава 15. Управление устройствами ввода.....291

ЧАСТЬ V
2D ГРАФИКА

Глава 16. Приложение Direct3D для 2D-графики.....308
Глава 17. Использование DirectDraw для рендеринга 2D-графики.....320

ЧАСТЬ VI
ДОБАВЛЕНИЕ СЕТЕВЫХ ВОЗМОЖНОСТЕЙ

Глава 18. Организация сети с равноправными узлами с помощью DirectPlay.....332
Глава 19. Создание сессии Client/Server.....350

Глава 20. Особенности более совершенного использования сетей.....366

Глава 21. Методы достижения максимального быстродействия.....379

ЧАСТЬ VII
ПРИЛОЖЕНИЯ

Приложение А. Использование сборок диагностики.....386

Приложение В. Воспроизведение музыки и видео.....389

Алфавитный указатель.....393

Оглавление

Предисловие **14**

Об авторе **15**

Посвящается **16**

Участники работы 16

Будем признательны за ваши отзывы и пожелания! **17**

Введение **18**

История Управляемого DirectX 18

Включенные пространства имен 20

Выбор программы 21

ЧАСТЬ I. ВВЕДЕНИЕ В КОМПЬЮТЕРНУЮ ГРАФИКУ 23

Глава 1. Введение в Direct3D **24**

Начало работы 24

Устройство Direct3D 25

Создание трехмерного треугольника 32

Автоматический сброс устройства во время изменения размеров окна 40

Создание освещения 41

Состояния устройства и преобразования 44

Свопинг-связки и рендеры 44

Краткие выводы 46

Глава 2. Выбор подходящего устройства **48**

Адаптеры системы 48

Поддержка аппаратного устройства 53

Проверка возможностей устройства 54

Краткие выводы 56

Глава 3. Введение в технологию рендеринга **57**

Использование вершинных буферов 57

Текстурирование объектов 68

Краткие выводы 74

Глава 4. Более совершенные технологии рендеринга **76**
Рендеринг с использованием примитивов различного типа 76
Использование индексных буферов 82
Использование буферов глубины или Z-буферов 87
Краткие выводы 90

Глава 5. Рендеринг «Mesh»-объектов **91**
Определение Mesh-объектов 91
Использование материалов и освещения 94
Использование Mesh-объектов для рендеринга сложных моделей 98
Краткие выводы 105

Глава 6. Использование Управляемого DirectX для программирования игр **106**
Выбор игры 106
Программирование игры 107
Добавление движущегося автомобиля в используемую сцену 118
Добавление препятствий 124
Последние штрихи 135
Краткие выводы 142

ЧАСТЬ П. ОСНОВНЫЕ КОНЦЕПЦИИ ПОСТРОЕНИЯ ГРАФИКИ **143**

Глава 7. Использование дополнительных свойств и возможностей Mesh-объектов **144**
Создание копий Mesh-объектов 144
Оптимизация данных Mesh-объекта 146
Упрощение существующих Mesh-объектов 149
Объединение вершин в Mesh-объектах 156
Создание нескольких объектов из одного 158
Краткие выводы 161

Глава 8. Введение в ресурсы **162**
Класс ресурсов 162
Использование вершинных и индексных буферов 163
Механизм Locking, блокировка используемых буферов 164
Управление процедурой блокировки 167
Использование текстурных ресурсов 169
Блокировка текстур и получение описаний 171
Краткие выводы 176

Глава 9. Применение других типов Mesh **177**
Упрощение Mesh-объектов 177
Управление степенью детализации, класс прогрессивных Meshes-объектов 181
Рендеринг патч-объектов. Тесселяция объектов 185
Примеры тесселированных объектов 190
Краткие выводы 192

Глава 10. Использование вспомогательных классов **193**
Рисование линий 193
Отображение текста 196
Рендеринг на поверхности 201
Рендеринг текстур Environment Maps 205
Краткие выводы 214

ЧАСТЬ III. БОЛЕЕ СОВЕРШЕННЫЕ МЕТОДЫ ПОСТРОЕНИЯ ГРАФИКИ **215**

Глава 11. Введение в программируемый конвейер, язык тендеров **216**
Рендеринг треугольника с использованием программируемого конвейера 216
Использование шейдеров для рендеринга, использование техник «TECHNIQUE» 225
Использование программируемого конвейера для рендеринга mesh-объектов 228
Использолвание языка HLSL для создания пиксельного шейдера 230
Краткие выводы 236

Глава 12. Использование языка шейдеров HLSL **237**
Использование простых формул для моделирования анимации ... 237
Объединение цветов текстуры с цветами поверхности 239
Текстуры освещения 244
Моделирование световых бликов 247
Краткие выводы 256

Глава 13. Рендеринг скелетной анимации **257**
Создание иерархической системы фреймов 257
Загрузка объектов с анимацией 262
Рендеринг анимированных объектов 267
Краткие выводы 274

ЧАСТЬ IV. ЗВУК И УСТРОЙСТВА ВВОДА **275**

Глава 14. Добавление звука **276**
Включение пространства имен SOUND 276
Загрузка и проигрывание статических звуков 276
Использование 3D-звука 281
Управление слушателем 285
Использование звуковых эффектов 287
Краткие выводы 290

Глава 15. Управление устройствами ввода **291**
Обнаружение устройств 291
Использование клавиатуры 294
Использование устройства мыши 299
Использование игровых клавиатур и джойстиков 301
Устройства обратной связи 303
Краткие выводы 306

ЧАСТЬ V. 2D ГРАФИКА **307**

Глава 16. Приложение Direct3D для 2D-графики **308**
Создание полноэкранного устройства отображения 308
Рендеринг спрайтов 310
Анимация спрайтов 315
Краткие выводы 319

Глава 17. Использование DirectDraw для рендеринга 2D-графики **320**
Создание полноэкранного устройства DirectDraw 320
Анимация спрайтов 327
Краткие выводы 330

ЧАСТЬ VI. ДОБАВЛЕНИЕ СЕТЕВЫХ ВОЗМОЖНОСТЕЙ **331**

Глава 18. Организация сети с равноправными узлами с помощью DirectPlay **332**
Адреса DirectPlay 332
Создание P2P-соединения 334
Начало сеанса 337
Использование модели событий 341
Работа в сети 344

Обработка потерянных сессий....347
Краткие выводы....349

Глава 19. Создание сессии Client/Server....350
Создание выделенного сервера....350
Создание соединения клиент-сервер....353
Отслеживание моментов присоединения и выхода из сети....357
Передача пакетов данных по сети....359
Формирование отклика клиента....363
Обработка отключения сервера....365
Краткие выводы....365

Глава 20. Особенности более совершенного использования сетей....-366
Модели событий и обработчики....366
Определение пропускной способности и статистики сети....368
Запуск приложений, использующих концепцию «Lobby»....371
Создание лобби-приложения....374
Добавление голосового чата....374
Краткие выводы....378

Глава 21. Методы достижения максимального быстродействия....379
Преобразование типов в объекты....379
Побочные эффекты моделей событий....380
Эффективность методов....383
Краткие выводы....384

ЧАСТЬ VII. ПРИЛОЖЕНИЯ....385

Приложение А. Использование сборок диагностики....386
Перечисление всех опций в системе....386
Проверка отдельных пунктов....388

Приложение В. Воспроизведение музыки и видео....389
Воспроизведение звукового файла....389
Воспроизведение видео файла в отдельном окне....390
Использование видео файла в качестве текстуры....390

АЛФАВИТНЫЙ УКАЗАТЕЛЬ....393

Введение

DirectX API является мощным программным интерфейсом, позволяющим разработчикам компьютерной графики писать быстродействующие приложения, используя стандартизированный набор интерфейсов. Первый выпуск DirectX совпал с выходом операционной системы Windows 95 и назывался «Games SDK» (средства SDK для разработки игр). С тех пор было выпущено более восьми новых версий API, однако, они предназначались только для разработчиков, использующих языки программирования C и C++, поэтому существовала целая группа разработчиков, которые не имели удобного доступа к функциональным возможностям DirectX.

История Управляемого DirectX

Первый выпуск Управляемого DirectX пришелся на 20 декабря 2002 (первая версия DirectX 9), однако проект продолжал находиться в разработке еще в течение некоторого времени. Позднее, во время разработки циклов для DirectX 8.1 и продолжению работы над DirectX для Visual Basic, мы начали прогон бета-версии Visual Studio.NET, и для нас сразу стала очевидна актуальность и перспективность данного направления. Я продолжил работу над прототипом «DirectX .NET», который впоследствии стал называться Управляемым DirectX.

Вначале предполагалось, что Управляемый DirectX будет столь же мощным, как и API. Ранее пользователи DirectX для Visual Basic не могли создавать приложения, сравнимые по качеству с приложениями DirectX API. Частично это было связано непосредственно со временем выполнения процедур на Visual Basic, а частично с нехваткой ресурсов библиотек DLL.

В DirectX 8 мы попробовали устранить некоторые из этих проблем, удалив интерфейсные уровни для большинства характеристик API, Direct3D. Вместо имеющихся proxi DLL библиотек, которые сортировали или маршализировали данные, мы переместили Direct3D API непосредственно в библиотеку типов (OLE library). Правда, это совсем незначительно улучшало характеристики, а также чрезмерно усложняло интерфейс API, особенно для разработчиков на Visual Basic, на которых мы ориентировались.

После выпуска DirectX 8 стало очевидно, что интерфейс API не совсем удобен в использовании. Примеры были сложны для понимания, и код не напоминал привычные приложения на Visual Basic.

Большинство разработчиков, использующих Visual Basic, нашли API слишком сложным в применении, а разработчики на C++ не видели причин переключаться на API, поскольку для них это было неудобно и невыгодно. До какого-то момента мы были интересны только домашним энтузиастам, а студии-разработчики компьютерных игр даже не знали о нашем существовании. Поэтому было необходимо направить работу над нашим новым API на достижение быстродействия, а также удобства его использования, что явилось бы достаточно вескими причинами для перехода из C++ кода, на котором писались многие программы, на C#. Таким образом, была намечена концепция для разработки Управляемого DirectX.

На Конференции Разработчиков Компьютерных Игр в Сан-Хосе в 2002 г. была представлена первая альфа-версия Управляемого DirectX на базе версии DirectX 8.1., написанная исключительно на C# .

В этой версии имелось большинство базовых компонентов DirectX, включая DirectMusic, который был исключен из окончательной версии Управляемого DirectX. Единственными компонентами DirectX, не представленными в первой альфа-версии, были DirectDraw и DirectShow, главным образом вследствие того, что мы не решили окончательно, включать эти компоненты или нет.

Оглядываясь назад, на первую версию, можно без сомнения предположить, что она была перспективной. Эта версия демонстрировала достаточную оперативность Управляемого DirectX, и отзывы, которые мы получили от пользователей, были на удивление положительны, хотя рабочие характеристики этого уровня были еще недостаточно высоки. В некоторых случаях мы смогли бы получить качество выполнения, соответствующее родному API, но даже при прокрутке простого сценария создания кубической текстуры версия с управляемым кодом существенно проигрывала в скорости по сравнению с версией на C++.

Мы потратили еще несколько месяцев на подготовку следующего варианта — первой бета-версии DirectX 9. Мы сосредоточили внимание на самых острых проблемах, добавили поддержку DirectDraw и удалили поддержку DirectMusic. Мы также включили в команду разработчиков, знакомых с C#. Полученная нами по результатам нашей работы обратная реакция была быстрой и однозначной — эксплуатационные характеристики были замечательные. Управляемый DirectX отличался от других уп-

равляемых API и напоминал скорее интерфейс COM API. Все компоненты среды .NET Runtime остались общими, тогда как библиотеки Управляемого DirectX полностью изменились. Перед данным проектом мы ставили две задачи: быстрота и простота в управлении. Стало очевидно, что мы не учли последнего.

Полная перестройка API началась сразу после выхода первой бета-версии. Мы обсуждали необходимые вопросы и с командой разработчиков среды .NET runtime, и с пользователями бета-версии. В результате мы собрали мнения всех заинтересованных источников и смогли спокойно выяснить и устранить недостатки данной версии. Я составил список параметров и требований, которых было необходимо придерживаться. Некоторые изменения были достаточно просты в осуществлении, например, константы или имена структур в DirectX (и COM вообще). Другие изменения были более сложными, в частности, добавление реестров к классам. Мы выпустили вторую бета-версию и с тревогой ждали обратную реакцию от наших пользователей.

К счастью, отзывы были положительными. Мы предприняли шаги в правильном направлении, и разработка интерфейса API продвигалась к реальному продукту.

За месяц до появления заключительной версии DirectX 9 один из вариантов бета-версии был разослан пользователям по почте. Это был не полный комплект SDK, но он включал в себя Управляемый DirectX API. Мы даже не представляли, как пользователи будут работать с этим без каких-либо образцов и документации. Это было серьезным испытанием для нас. Обратная реакция, которую мы получили, была положительной. Люди отправляли на соответствующие форумы и Web-сайты статьи, описывающие, как использовать API.

Когда в декабре вышла заключительная версия Управляемого DirectX 9, стало ясно, что мы добились желаемого результата.

Включенные пространства имен

Управляемый DirectX API имеет достаточно большой список имен, namespaces, разделяющих функциональные возможности различных компонентов DirectX. Имеется также универсальное пространство имен «Microsoft.DirectX», которое размещает базовые возможности. Полный список пространства имен, включенных в Управляемый DirectX, перечислен в таблице 1.

Таблица 1. Пространство имен Управляемого DirectX

Microsoft.DirectX	Корневой каталог, содержит все общие коды
Microsoft.DirectX.Direct3D	Direct3D API графика, так же как вспомогательная библиотека D3DX
Microsoft.DirectX.DirectDraw	Графика DirectDraw API
Microsoft.DirectX.DirectPlay	Сетевой DirectPlay API.
Microsoft.DirectX.DirectSound	Аудио DirectSound API.
Microsoft.DirectX.Directlnput	Пользовательский вход в Directlnput API
Microsoft.DirectX. AudioVideoPlayback	API простого звукового и видео воспроизведения
Microsoft.DirectX.Diagnostics	Простая диагностика API
Microsoft.DirectX.Security	Контроль доступа на глубокие уровни DirectX
Microsoft.DirectX.Security.Permissions	Классы разрешения для защиты доступа в DirectX

Как видно, этот список охватывает большинство функциональных возможностей, включенных в DirectX. В этой книге мы подробно рассмотрим приложения, связанные с пространством имен в Direct3D, и коснемся некоторых из перечисленных приложений.

Выбор программы

Прежде чем начать описание работы с Управляемым DirectX, мы хотели бы обратить внимание на некоторые моменты.

Первое, что нам понадобится для работы, — редактор исходного кода и среда разработки. Я бы рекомендовал Visual Studio.NET 2003, поддерживающий объекты Microsoft. Независимо от типа редактора, понадобится версия 1.1 среды .NET runtime, которую вы можете установить с прилагаемого CD диска.

Также понадобится установленный DirectX 9.0 SDK Update. Я бы рекомендовал DirectX 9.0 Software Development Kit Update, включенный в указанный CD диск. Там же можно найти множество примеров, а также документацию на Управляемый DirectX.

Графические объекты, описываемые в первых разделах книги, должны запускаться и работать на любой современной графической карте. Однако, более сложные объекты требуют установки более совершенных видеокарт, способных работать с программами текстурирования, с вершинными и пиксельными шейдерами. Карта GeForce3 является подходящей для таких задач, однако, я бы порекомендовал карту, способную работать с шейдером модели 2.0 (например, Radeon 9700 и выше).

Описывая принципы программирования Управляемого DirectX, мы подразумеваем, что читатель знаком с данной тематикой. В противном случае, книга может быть достаточно сложна для восприятия. Книга предназначена для разработчиков компьютерной графики, имеющих навыки и опыт работы с DirectX, которые могут найти здесь дополнительную информацию относительно создания более продвинутых мультимедийных приложений, использующих Управляемый DirectX. Код и алгоритмы, приводимые в тексте этой книги, написаны на языке C#, но на компакт-диске могут быть приведены версии процедур, написанные на Visual Basic.NET. Компакт диск содержит и другие исходники программ, как на языке C#, так и на Visual Basic.NET. Также DirectX 9 SDK Update можно загрузить с сайта http://msdn.microsoft.com.

Теперь мы можем приступить к рассмотрению принципов работы и программирования с помощью Управляемого DirectX.

ЧАСТЬ I

ВВЕДЕНИЕ В КОМПЬЮТЕРНУЮ ГРАФИКУ

Глава 1. Введение в Direct3D

Глава 2. Выбор подходящего устройства

Глава 3. Введение в технологию рендеринга

Глава 4. Более совершенные технологии рендеринга

Глава 5. Рендеринг «Mesh»-объектов

Глава 6. Использование Управляемого DirectX для программирования игр

Глава 1. Введение в Direct3D

Добавление трехмерной графики является одним из наиболее популярных аспектов мультимедийного приложения. Используя современные мощные системы обработки и графический процессор GPU, можно отображать самые реалистичные сцены в реальном масштабе времени. Управляемый Direct3D позволяет разработчикам компьютерных игр просто и быстро формировать на экране сложные элементы графики и мультипликации. В этой главе мы охватим следующие моменты.

- Создание Direct3D устройств.
- Рендеринг или визуализация изображения в различных координатных системах (экранные и обычные координаты).
  Добавление освещения к отображаемому полигону.

Начало работы

Перед запуском своего первого приложения в среде Managed Direct3D необходимо сделать следующее:

1. Загрузить приложение Visual Studio.NET и создать новый проект (new project).
2. Выбрать в качестве пространства проекта или программы среду Visual C#, создать новое окно приложений (понадобится также место для рендеринга и стандартных форм Windows).
3. Придумать и записать имя нового проекта.

После того, как мы создали новый проект, необходимо убедиться, что к проекту добавлены необходимые ссылки на Управляемый DirectX, что позволит в дальнейшем использовать его компоненты. Для этого, используя меню проекта, кликните на команду добавления **Add References** и добавьте приложения **MicrosoftDirectX** и **Microsoft.DirectX.Direct3D.** Пока это все, что может понадобиться нам на первом этапе.

Теперь, прежде чем начать использовать Управляемый DirectX, необходимо добавить два новых элемента в директиву «using», позволяющих автоматически идентифицировать используемые нами компоненты. Это можно сделать, открыв окно записи кода для главной формы Windows в вашем приложении (по умолчанию forml.cs) и добавив следующие строки в конце операторов «using»:

```
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;
```

Последние описанные действия являются необязательными. Просто они позволяют не классифицировать каждый раз объекты перед их ис-

пользованием. Итак, мы готовы к написанию первого приложения с помощью Управляемого DirectX.

Устройство Direct3D

Выполнение всех графических операций в Direct3D определяется классом устройства. Можно полагать, что этот класс аналогичен графическому устройству и графической карте, установленной на компьютере.

Устройство является первостепенным относительно всех других графических объектов в сценарии. Компьютер может иметь или не иметь графические устройства, при этом, используя приложение Direct3D, можно управлять работой некоторых или большинства из них.

Существуют три типа конструктора программ, которые могут использоваться для создания устройства. Пока мы будем использовать только один из них, а остальные рассмотрим позднее. Данный конструктор включает следующие параметры:

```
public Device (System.Int32 adapter, Microsoft.DirectX.Direct3D.DeviceType
    deviceType,  System.Windows.Forms.Control  renderWindow,
    Microsoft.DirectX.Direct3D.CreateFlags  behaviorFlags,
    Microsoft.DirectX.Direct3D.PresentParameters presentationParameters )
```

Первый параметр *adapter* (адаптер) относится к тому физическому устройству, которое мы хотим классифицировать.

Каждое устройство в компьютере имеет идентификатор адаптера (от О до числа, на единицу меньшего, чем число имеющихся устройств). Нулевое значение для параметра adapter подразумевает установку устройства по умолчанию.

ПЕРЕГРУЗКИ ОПЕРАЦИЙ В КОНСТРУКТОРЕ УСТРОЙСТВ

Вторая перегрузка для устройства идентична первой, за исключением параметра *renderWindow,* который принимает указатель IntPtr для неуправляемого окна. Последняя перегрузка принимает отдельный указатель IntPtr, который является неуправляемым COM указателем для интерфейса IDIRECT3DDEVICE9. Она может использоваться при необходимости работы с неуправляемым приложением посредством управляемого кода.

Следующий параметр *DeviceType* (тип устройства) сообщает приложению Direct3D, какой тип устройства мы хотим создать. Обычно, значением этого параметра является опция DeviceType.Hardware, подразумевающая использование аппаратного устройства. Другой вариант значе-

ния — опция DeviceType.Reference, которая позволяет использовать дополнительное или эмулированное устройство растеризации (rasterizer), реализующее все возможности Direct3D Realtime, но работающее чрезвычайно медленно. Эту опцию целесообразно использовать главным образом для отладки программных задач или для проверки параметров приложений, не поддерживаемых вашей видеокартой.

ИСПОЛЬЗОВАНИЕ ПРОГРАММНЫХ УСТРОЙСТВ

Следует обратить внимание, что эмулированный rasterizer поставляется только с DirectX SDK, при использовании DirectX Runtime эта опция будет недоступна. Последнее значение — DeviceType.Software позволяет использовать программный rasterizer.

Параметр *renderWindow* (окно рендеринга) связывает окно с созданным устройством. Поскольку класс управления окнами содержит дескриптор окна, это облегчит использование соответствующего класса в качестве окна рендеринга. Для этого параметра можно использовать форму, панель, или любой другой производный класс. На данном этапе мы будем использовать только формы.

Следующий параметр *behaviorFlags* (флажки режимов) используется для управления режимами устройства после его создания. Большинство значений списка CreateFlags может быть объединено, что позволит при необходимости установить множественные режимы работы устройства. Некоторые из этих флажков являются взаимоисключающими, однако, об этом будет рассказано позже. Пока мы будем использовать флажок SoftwareVertexProcessing, передающий всю обработку вершин центральному процессору. Несмотря на то, что таким образом операции выполняются медленнее, чем при использовании графического процессора, надежнее будет использовать именно такой подход, т.к. не всегда известно наверняка, поддерживает ли наша графическая плата этот режим.

Последний параметр *presentationParameters* управляет представлением данных на экране. Посредством этого класса можно управлять любыми параметрами отображаемых данных. Позже мы коснемся деталей этой структуры, а сейчас уделим внимание значениям *Windowed* и *SwapEjfect.*

Значение *Windowed* является логической переменной, определяющей, работает ли устройство в полноэкранном (значение <false»-«ложь»), или в оконном режиме (значение «true»-«истина»).

Значение *SwapEffect* используется для описания режима работы буферной подкачки. Если вы выбрали значение SwapEffectFlip, будет создан вторичный буфер и произойдет копирование, независимо от состояния первичного буфера на данный момент. Значение SwapEffect.Copy сходно со значением Flip, но потребует от программиста установки зна-

чения «1» для числа вторичных буферов. В данном случае мы выберем опцию SwapEffect.Discard, которая просто сбрасывает содержимое буфера, если он не готов к представлению.

Теперь, когда у нас имеется эта информация, мы можем создать устройство, вернувшись к нашему коду. Сначала мы должны определить объект, используемый нашим приложением, и добавить к нему новую переменную (private variable) «device». Для этого включите следующую строку в раздел определения класса:

```
private Device device = null;
```

Теперь мы добавляем к указанному классу новую функцию InitializeGraphics, используя описанный выше конструктор программ. Добавьте следующий код инициализации:

```
/// <summary>
/// We will initialize our graphics device here
/// </summary>
public void InitializeGraphics))
(
   // Set our presentation parameters
   PresentParameters presentParams = new PresentParameters()
   presentParams.Windowed = true;
   presentParams.SwapEffeet = SwapEffect.Discard;
   // Create our device
   device = new Device(0, DeviceType.Hardware, this,
      CreateFlags.SoftwareVertexProcessing, presentParams);
1
```

Таким образом, мы определяем параметр предоставления presentParams, в котором устанавливаем описанные выше значения (Windowed и SwapEffect), и затем создаем устройство. Нулевое значение параметра adapter подразумевает установку устройства по умолчанию. В результате мы создали фактическое аппаратное устройство, в противоположность устройству, эмулированному программными средствами.

Следует обратить внимание, что в рассмотренном коде использовалось ключевое слово «this», подразумевающее наше текущее окно рендеринга. Поскольку наше приложение, и в особенности этот класс, являются своего рода винформой, мы просто используем его. Как уже упоминалось, мы возложили вершинную обработку на наш центральный процессор.

Теперь осталось переписать основной метод для вызова процедуры инициализации InitializeGraphics:

```
static void Main()
{
   using (Form1 frm = new Form1())
   {
      // Show our form and initialize our graphics engine
      frm.Show();
      frm.InitializeGraphics();
      Application.Run(frm);
   }
}
```

Что мы здесь сделали? Сначала, мы добавили оператор или директиву. «using» для создаваемой формы. Это гарантирует размещение нашей формы в каталоге, когда приложение оставляет область этого блока. Затем, мы добавили команду Show, для того чтобы гарантировать, что окно загружено и отображено (и таким образом создан хэндл окна) прежде, чем мы попытаемся создать наше устройство. Затем мы вызвали функцию инициализации устройства и использовали стандартный алгоритм для нашего приложения. Теперь мы можем скомпилировать это первое приложение в Direct3D и запустить его.

Следует добавить, что созданное приложение довольно простое. Оно создает устройство, но фактически ничего с ним не делает, и глядя на это выполняемое приложение, мы не можем сказать, что оно как-то отличается от первоначально созданного пустого C# проекта. Необходимо переписать его и отобразить что-нибудь на нашем экране.

Классы Windows-form имеют встроенный метод, позволяющий отслеживать момент, когда необходимо обновить или «перерисовать» окно — это метод OnPaint (можно также использовать приложение Paint). Добавим этот метод в наш код рендеринга. Нет смысла делать сейчас какие-то сложные вещи, попробуем просто очистить окно и заполнить его чистым цветом. Добавьте следующий код к определению класса:

```
protected override void OnPaint(System.Windows.Forms.PaintEventArgs e)
{
   device.Clear(ClearFlags.Target, System.Drawing.Color.CornflowerBlue,
   1.0f, 0);
   device.Present();
}
```

Для того чтобы заполнить наше окно чистым цветом, воспользуемся процедурой Clear. В этом случае мы используем один из предварительно определенных цветов, а именно CornflowerBlue. Первый параметр метода Clear определяет, что именно мы хотим очистить, в данном примере это будет текущее окно.

Вторым параметром является цвет, используемый при очистке нашего окна. После того как устройство очищено, мы можем обновить состояние дисплея. Данную процедуру будет выполнять метод представления Present. Существует несколько различных вариантов данного метода, один из них, показанный здесь, использует всю область устройства. Другие перегрузки мы обсудим позже.

Запустив приложение, мы увидим окно, заполненное синим цветом (цвет василька). Можно изменять размеры окна или разворачивать его на весь экран, и каждый раз окно будет заполняться синим цветом. Несмотря на то, что наше новое устройство уже что-то из себя представляет, созданное приложение все еще довольно скучно. Можно было сделать то же самое, просто установив цвет фона формы в конструкторе программы. Теперь попробуем нарисовать еще что-нибудь.

Основным объектом, используемым в трехмерной графике, является треугольник, он же является элементарным полигоном при построении изображений. С достаточным количеством треугольников можно представить и отобразить что угодно, даже гладкие изогнутые поверхности. Таким образом, в качестве нашего первого рисунка можно выбрать треугольник. Чтобы упростить процедуру, будем рисовать треугольник, используя экранные координаты. Для этого нам необходимо, во-первых, сформировать данные, включающие информацию о треугольнике, во-вторых, дать команду устройству для выполнения процедуры рисования, другими словами, вызвать метод рисования.

К счастью, Управляемый DirectX уже имеет встроенную конструкцию для хранения и обработки информации об указанных треугольниках. В составе имен Direct3D существует класс вершин CustomVertex, который размещает используемые в Direct3D различные сборки вершин в формате «CustomVertex». Структура вершинного формата содержит данные в понятном для Direct3D виде. Мы коснемся этого вопроса позже, а сейчас будем использовать для нашего треугольника структуру TransformedColored — преобразование цветов. Эта структура сообщает Direct3D runtime, что наш треугольник не требует каких-либо преобразований, например, вращения или перемещения, поскольку мы будем определять или устанавливать его положение посредством экранных координат. Данная структура включает цветовые параметры для каждой вершины треугольника. Вернемся к методу OnPaint и добавим следующий код после вызова Clear:

```
CustomVertex.TransformedColored[j  verts  =  new
CustomVertex.TransformedColored[3];
verts[OJ.SetPosition(new Vector4 (this. Width / 2. Of, 50. Of, 0.5f, 1.0 f).) ;
verts[0].Color  =  System.Drawing.Color.Aqua.ToArgb();
verts[lJ.SetPosition(new  Vector4(this.Width  -  (this.Width  /  5.Of),
   this.Height  -
    (this.Height  /  5.Of),  0.5f,  1.Of));
```

```
verts [I] .Color = System.Drawing.Color.Black.ToArgb();
verts[2].SetPosition(new Vector4(this.Width / 5.Of, this.Height -
    (this.Height / 5.Of)
    , 0.5f, 1.Of));
verts [2] .Color = System.Drawing.Color.Purple.ToArgb();
```

Так как каждый элемент массива, который мы создали, представляет одну точку в нашем треугольнике, таким образом будет создано три элемента. Затем, используя структуру Vector4, мы вызываем метод SetPosition для каждого элемента массива. Местоположение преобразованной вершины определяется значениями X и Y в координатах экрана (относительно начала координат окна 0,0), а также значениями глубины Z и rhw- (обратная величина «homogenous w»). Последние два значения для нашей выборки игнорируются. Структура Vector4 очень удобна для задания информации о треугольнике.

Далее мы устанавливаем цвет каждой вершины. Следует обратить внимание на то, что мы должны вызвать используемый для этого метод задания стандартных цветов ToArgb. В приложении Direct3D подразумевается, что цвета будут определяться 32-разрядными целыми числами, и данный алгоритм преобразует базовый цвет в этот формат.

Важно помнить и то, что для определения координат нашего треугольника мы используем ширину и высоту текущего окна и должны это учитывать при изменении размеров треугольника.

Теперь, когда мы определили данные треугольника, необходимо сообщить приложению Direct3D о том, что мы готовы начать рисование, и указать, каким образом мы будем это делать. Для этого можно использовать следующий код, добавляющийся в процедуру OnPaint:

```
device.BeginScene ();
device.VertexFormat = CustomVertex.TransformedColored.Format;
device.DrawUserPrimitives(PrimitiveType.TriangleList, 1, verts);
device.EndScenef);
```

Здесь команда BeginScene уведомляет Direct3D о том, что мы собираемся что-то нарисовать, и определяет его «готовность к этому». Далее, мы должны сообщить, что именно мы собираемся нарисовать. Метод VertexFormat устанавливает для приложения Direct3D runtime используемый нами в дальнейшем формат непрограммируемого конвейера (формат *«fixed function pipeline»)*. В данном случае для рисования мы используем преобразованный и цветной вершинный формат. О непрограммируемом конвейере мы расскажем позднее.

Функция или метод DrawUserPrimitives определяет место, где размещается рисунок. Что же означают параметры этой функции? Первый параметр — это тип примитива, который мы планируем рисовать. Суще-

ствуют различные примитивы, которые нам доступны, но сейчас, для простоты, мы будем рисовать набор треугольников TriangleList. Второй параметр — число треугольников, которые мы хотим нарисовать, или другими словами, общее число вершин, разделенное на три. Для случая одного треугольника, это значение равно «1». Последний параметр этой функции означает, какие данные Direct3D будет использовать при рисовании. В нашем случае это вершины.

Последняя команда EndScene сообщает приложению Direct3D runtime об окончании процедуры рисования. Эта команда должна присутствовать всегда, если использовалась команда BeginScene.

Теперь можно компилировать и запускать наше новое приложение. Обратите внимание, что на цветном фоне теперь нарисован треугольник. Цвета вершин треугольника задаются нами в программе, а на изображении цвета внутри треугольника плавно изменяются от одного к другому. Direct3D автоматически интерполирует цвета. Для практики попробуйте изменить базовые цвета и почувствуйте результат.

Если вы уже опробовали работу нового приложения, то, наверное, заметили некоторые особенности.

Например, при уменьшении размера окна его содержимое не обновляется, и никаких действий не происходит. Причина в том, что Windows не рассматривает уменьшение размера окна как случай, когда вам необходимо перерисовать целое окно. В конце концов, вы просто удалили часть информации, которая была отображена, при этом вы не стерли данные, который там находились.

К счастью, есть простой способ обойти эту особенность. Мы можем сообщить Windows, что нам всегда необходимо окно, цвет которого будет обновляться. Это легко реализовать, аннулируя окно в конце метода OnPaint.

```
this.Invalidate();
```

Может показаться, что мы сломали наше приложение! В результате выполнения этой командной строки мы видим главным образом пустой экран, и наш мерцающий треугольник. И этот эффект проявляется более отчетливо, когда вы изменяете размеры вашего окна. Что же произошло? Оказывается, Windows пытается обработать ситуацию и перерисовывает текущую форму нашего окна после аннулирования, окрашивая и внешнюю сторону объекта. Его легко зафиксировать, изменяя параметр style созданного нами окна. Для этого в конструкторе формы замените сегмент «TODO» следующей строкой:

```
this.SetStyLe(ControlStyles.ALlPaintinglnWmPaint | ControlStyles.Opaque,
true);
```

Теперь, когда мы запускаем приложение, все работает нормально. Подразумевается, что при выполнении этого приложения полное закрашивание должно происходить внутри объекта (сообщение WmPaint класса Win32), и что наше окно будет не прозрачно. Это гарантирует, что никакого бесконтрольного закрашивания вне пространства нашего окна не произойдет. Обратите внимание, что если при изменении размеров окно попадает в невидимую область, приложение выполняться не будет.

Создание трехмерного треугольника

Созданное нами в предыдущем параграфе приложение не является трехмерным. Оно касается только плоского цветного треугольника, нарисованного внутри окна, которое можно легко создать с помощью графического интерфейса GDI. Теперь возникает вопрос, как можно создать реалистичный объект в трехмерном пространстве? Для этого нужно изменить наше существующее приложение.

Если вспомнить, что раньше, когда мы формировали данные нашего треугольника, мы использовали нечто, называемое преобразованными координатами *(transformed* coordinates). Эти координаты, как известно, лежат в пространстве экрана и легко определяются. А что если использовать непреобразованные координаты? Они и составляют наибольшую часть сцены в современной 3D-игре.

Когда мы определяем эти координаты, нам необходимо позиционировать каждую вершину не в экранном пространстве, а в пространстве мировых координат. Можно представить внешнее пространство, как бесконечное трехмерное Декартово пространство, в любом месте которого мы можем размещать объекты. Теперь давайте модифицируем наше приложение и нарисуем непреобразованный треугольник.

Вначале мы изменим данные нашего треугольника, чтобы использовать один из типов непреобразованных вершинных форматов. В этом случае нас волнует местоположение и цвет нашей вершины, поэтому мы выбираем команду CustomVertex.PositionColored. Замените ваш код для данных треугольника на следующий:

```
CustomVertex.PositionColored[] verts = new CustomVertex.PositionColored[3];
verts[0].SetPosition(new Vector3(0.0f, 1.0f, 1.0f));
verts[0].Color = System.Drawing.Color.Aqua.ToArgb();
verts[1].SetPosition(new Vector3(-1.0f, -1.0f, 1.0f));
verts[1].Color = System.Drawing.Color.Black.ToArgb();
verts[2].SetPosition(new Vector3(1.0f, -1.0f, 1.0f));
verts [2] .Color = System.Drawing.Color.Purple.ToArgb();
```

Измените также свойства вершинного формата VertexFormat:

```
device.VertexFormat = CustomVertex.PositionColored.Format;
```

Теперь, если вы запустите приложение, ничего не произойдет, и вы вернетесь назад, к вашему цветному экрану. Прежде, чем выяснить причину, давайте займем одну минуту, чтобы описать последние действия. Легко заметить, что мы переписали наши данные для использования вместо них структуры PositionColored. Эта структура будет сохранять вершину в пространстве внешних координат, а также цвет каждой вершины.

Поскольку вершины не являются преобразованными, мы используем класс Vector3 вместо класса Vector4, который мы использовали с преобразованным объектом.

Элементы Vector3 структурируют карту непосредственно в координатах мирового пространства (координаты X, Y, и Z). Мы также должны удостовериться, что приложению Direct3D известно о том, что мы изменили тип рисуемых данных. Таким образом, чтобы использовать новую непреобразованную и цветную вершину, мы изменяем наш непрограммируемый конвейер («fixed function pipeline»), модифицируя свойства вершинного формата VertexFormat.

Почему же ничего не отображается, когда мы запускаем наше приложение? Проблема в том, что, располагая нашу вершину в мировом пространстве, мы не дали Direct3D никакой информации относительно того, как он должен ее отобразить. Мы должны добавить в сцену камеру, которая может определять и рассматривать нашу вершину. В преобразованных координатах камера была не нужна, потому что размещение вершины в экранном пространстве однозначно определяло ее местоположение.

Камера управляется с помощью двух различных преобразований. Каждое преобразование определяется матрицей 4x4, которую вы можете переслать в Direct3D. Преобразование проекции используется для определения того, как сцена проецируется на экран. Один из самых простых способов создания матрицы проекции состоит в использовании функции PerspectiveFovLH в классе Matrix.

Таким образом, в результате выполнения описанных процедур создается матрица проекции левой системы координат.

Что означает левая система координат, и какое это имеет значение? В обычной Декартовой системе координат положительная ось X направлена слева направо, тогда как положительная ось Y направлена вверх. Третьей координатой является ось Z. В левой системе координат положительная ось Z направлена от вас, в то время как в правой системе координат, положительная ось Z направлена к вам. Это легко запомнить, если направить пальцы руки по направлению оси X, а большой палец — по направлению оси Y, как показано на **рис.** 1.1. Другими словами, если, глядя из конца вектора Z, поворот от оси X к оси Y осуществляется про-

: Зак. 604

тив часовой стрелки — это правая система координат, если по часовой стрелке — левая система координат.

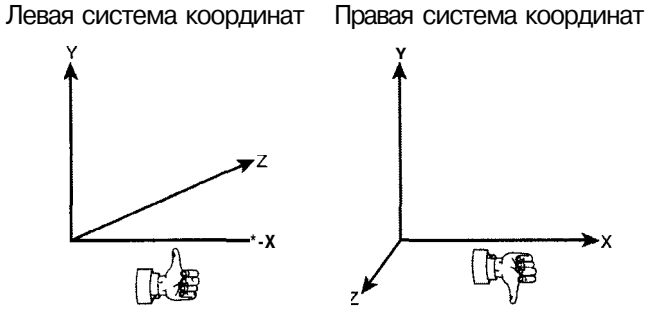


Рис. 1.1. Трехмерная система координат

Приложение Direct3D использует левую систему координат. Если вы пишете код для правой системы координат, необходимо сделать две вещи. Сначала нужно перестроить порядок треугольников таким образом, чтобы они расположились по часовой стрелке от передней стороны (вскоре мы объясним данную процедуру). Затем использовать матрицу масштабирования внешнего пространства относительно отрицательной оси Z. вы можете сделать это путем перестановки членов M31, M32, M33 и M34 матрицы вида. Затем вы можете использовать версию RH матричной функции для построения матрицы правой системы.

Теперь, создавая новое приложение, мы будем использовать только левую систему координат, с которой работает Direct3D. Пример для нашей функции матрицы проекции имеет вид:

```
public static Microsoft.DirectX.Matrix PerspectiveFovLH ( System.Single
   fieldOfViewY,
System.Single aspectRatio , System.Single znearPlane , System.Single
   zfarPlane )
```

Преобразование проекции используется для описания сцены, представленной в виде усеченной пирамиды *(frustum),* где внутренняя часть пирамиды является просматриваемой областью нашей сцены. Два параметра в нашей функции, ближняя и дальняя плоскости *(znearPlane* и *zfarPlane)* описывают пределы этой пирамиды, причем дальняя плоскость является основанием, а ближняя плоскость проходит по месту отсечения вершины пирамиды, **рис.1.2.** Параметр поля зрения определяется углом при вершине пирамиды, **рис.1.3,** а форматное соотношение сторон — аналогично форматам телевидения, например, широкоформатное телевидение имеет соотношение 1.85. Можно понимать этот параметр как отношение ширины изображения к высоте. Следует отметить, что

Direct3D прорисовывает только те объекты, которые заключены внутри усеченной пирамиды.

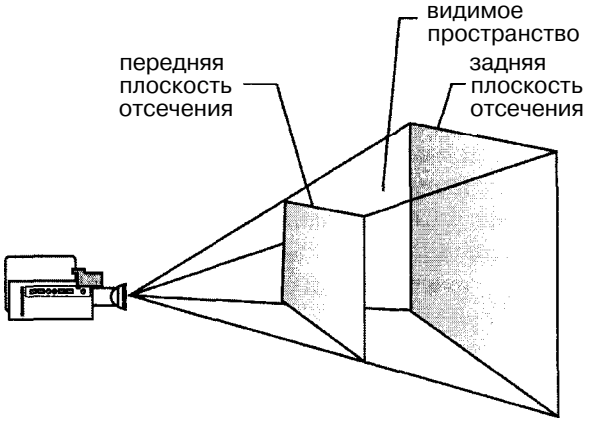


Рис. 1.2. Визуализируемое видимое пространство

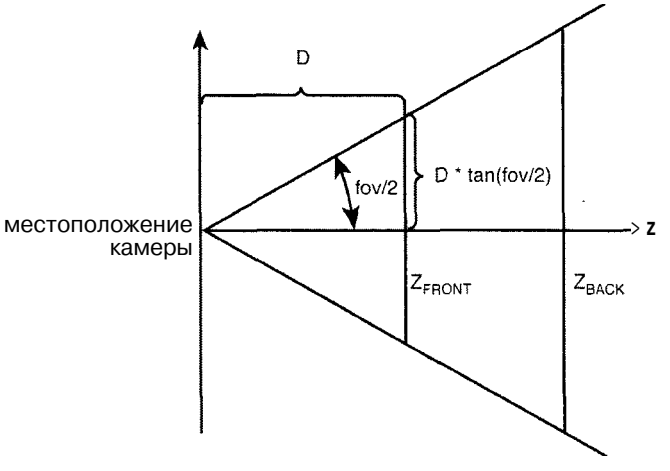


Рис. 1.3. Определение поля зрения

Итак, теперь попробуем добавить камеру к нашей сцене, поскольку без камеры мы не сможем выполнять описанные выше преобразования:

```
public static Microsoft.DirectX.Matrix LookAtLH(Microsoft.DirectX.Vector3
    cameraPosition,
Microsoft.DirectX.Vector3 cameraTarget , Microsoft.DirectX.Vector3
    cameraUpVector )
```

Эта функция достаточно очевидна. Требуются три параметра, которые описывают свойства камеры. Первый параметр — положение камеры во внешней системе координат. Следующий параметр — местоположение того объекта во внешней системе координат, на который должна смотреть камера. Последний параметр — направление камеры, в нашем случае «Up» — вверх.

С описанием преобразования проекции и преобразования вида, Direct3D теперь имеет достаточно информации, чтобы отобразить наш недавно созданный треугольник. Давайте изменим наш код, чтобы отобразить некий объект. Мы добавим новую функцию SetupCamera (установка камеры) в нашу процедуру OnPaint сразу после вызова очистки Clear. Тело этой функции будет включать следующее:

```
private void SetupCamera()

    device.Transform.Projection = Matrix.PerspectiveFovLHl(float)Math.PI / 4,
        this.Width / this.Height, i.Of, 100.Of);
    device.Transform.View = Matrix.LookAtLH(new Vector3(0,3, 5.Of), new
    Vector3(),
        new Vector3(0,1,0));
i
```

Как вы можете видеть, мы создаем матрицу преобразования проекции для нашего устройства, чтобы обозначить приложению Direct3D область наблюдения. Затем мы определяем нашу матрицу вида и задаем информацию о положении камеры. Добавьте обращение к этой функции в нашу процедуру OnPaint после вызова очистки Clear и запустите еще раз наше приложение.

Что мы видим теперь? Мы нарисовали наш треугольник, но теперь он оказался весь черный, даже при том, что мы определили цвета этого треугольника. Еще раз, проблема заключается в различии между предварительно преобразованным треугольником (pretransformed), который мы нарисовали в первом приложении, и нашим непреобразованным треугольником теперь. В непреобразованной среде приложение Direct3D по умолчанию использует освещение, чтобы определить конечный цвет пиксела каждого примитива в сцене. Так как мы не используем никакого освещения в нашей сцене, нет никакой внешней засветки на нашем треугольнике, и поэтому он кажется черным. Если мы уже определили цвет и хотим, чтобы наш треугольник появился, можно пока просто выключить освещение в сцене, вы можете выполнить это, добавив следующую строку в конце вызова функции SetupCamera:

```
device.RenderState.Lighting = false;
```

ИСПОЛЬЗОВАНИЕ СОСТОЯНИЙ РЕНДЕРИНГА В УСТРОЙСТВЕ

Существует много состояний рендеринга, которые могут использоваться для управления различными сценами в конвейере рендеринга. Мы обсудим часть из них в последующих главах.

Теперь при запуске приложения мы увидим треугольник, напоминающий наш первый, предварительно преобразованный треугольник. Может показаться, что, выполнив немалое количество действий, мы вернулись в ту же самую точку, где мы находились еще до перехода к нашим непреобразованным треугольникам. Какие реальные преимущества мы получили от проведения этих изменений? Главное заключается в том, что теперь мы имеем треугольник в реальном трехмерном пространстве, это предпочтительнее, чем просто рисунок в координатах экрана.

Для более значимого восприятия того, что мы действительно имеем трехмерное изображение треугольника, попробуем вращать его. Как же это можно сделать? Это достаточно просто, мы должны изменить преобразование пространства.

Преобразование пространства используется в устройстве для преобразования объектов, нарисованных в пространстве моделей, где каждая вершина определена относительно модели, расположенной во внешнем пространстве. Преобразованием пространства может быть любая комбинация перемещения, вращения и масштабирования. Так как на данном этапе мы хотим просто вращать наш треугольник, то для этого мы выполним единственное преобразование. Каждое преобразование характеризуется своей матрицей. У матричного объекта может быть множество функций, позволяющих выполнять эти преобразования. Добавьте следующую строку к вашей функции SetupCamera:

```
device.Transform.World - Matrix.Rotations((float)Math.PI / 6.0f);
```

Эта строка сообщает приложению Direct3D о типе используемого преобразования для каждого рисуемого объекта. В данном случае это вращение нашего объекта относительно оси Z. Угол поворота задается строго в радианах. В библиотеке функций Direct3DX, которую мы добавим позже к нашему проекту, существует вспомогательная функция Geornetry.DegreeToRadians.

Пока мы выбрали произвольный угол вращения, чтобы просто показать сам эффект. В результате мы видим на экране наш треугольник, повернутый вокруг оси Z.

Теперь попробуем вращать треугольник непрерывно.

Для этого произведем следующее преобразование пространства:

```
device.Transform.World = Matrix.RotationZ((System.Environment.TickCount
    / 450.0f)
    / (float)Math.PI);
```

В результате этого мы видим треугольник, медленно вращающийся вокруг оси Z.

Движение кажется немного прерывистым, но это вызвано дискретностью параметра TickCount. Данный параметр в системном классе представляет из себя некоторый дополнительный интерфейс для метода GetTickCount в интерфейсе приложений Win32 API, который имеет врем менную разрешающую способность приблизительно 15 миллисекунд. Это означает, что значение, приведенное здесь, изменяется с приращением приблизительно 15 мс, что и вызывает это прерывистое поведение. Мы легко можем сгладить вращение при наличии нашего собственного счетчика, использующего свое приращение, отличное от TickCount. Добавьте новое значение переменной угла в виде числа с плавающей точкой. Тогда изменение вашего преобразования будет иметь следующий вид:

```
device.Transform.World = Matrix.RotationZ(angle / (float)Math.PI);
angle += 0.1f;
```

Теперь треугольник вращается более плавно. Управлять вращением (или другим перемещением) в этом случае не рекомендуется, так как приращение переменной зависит от скорости изменения кода рендеринга. С ростом быстродействия современных компьютеров, привязывая ваш код рендеринга к переменной, можно получить приложение, работающее либо чрезвычайно быстро на современных аппаратных средствах, либо слишком медленно на старых аппаратных средствах. Позже мы обсудим более эффективный способ привязки нашего кода к таймеру с гораздо большим временным разрешением.

Итак, наш вращающийся треугольник все еще не так реалистичен, как хотелось бы. Давайте попробуем вращать его одновременно вокруг других осей. К счастью, мы имеем необходимую для этого функцию.

Измените процедуру преобразование пространства следующим образом:

```
device.Transform.World = Matrix.RotationAxis(new Vector3(angle /
((float)Math.PI * 2.0f),
angle / ((float)Math.PI * 4.0f), angle / ((float)Math.PI * 6.0f)),
angle / (float)Math.PI);
```

Главное различие между этой строкой программы и предыдущей — это новая вызываемая функция RotationAxis. При вызове этой функции мы сначала определяем ось, вокруг которой мы хотим вращать треугольник, и угол вращения, используя простую математическую формулу для каждой оси. Теперь можно запустить это новое приложение и увидеть результат нашего преобразования.

Увы, теперь наш треугольник начинает вращаться вокруг своих осей, постоянно исчезая на мгновение, а затем вновь появляясь. Однако, мы уже сталкивались с похожей ситуацией, связанной с освещением невидимых поверхностей.

При отображении фигур отдельные поверхности или стороны объекта оказываются вне поля зрения камеры и не прорисовываются. Такой процесс называется *back face culling,* другими словами, устранение невидимых поверхностей (при двумерном изображении трёхмерных объектов). Теперь для приложения runtime осталось определить, насколько и какая часть поверхности попадает в поле зрения камеры. Быстрый взгляд на опции culling в Direct3D дает хорошую подсказку. Имеются в наличии три опции culling: none, clockwise и counterclockwise (соответственно: никак, по часовой стрелке и против часовой стрелки). В случае выбора варианта по часовой стрелке или против часовой стрелки, исходные объекты, чьи вершины расположены в противоположном порядке, не рисуются.

Глядя на наш треугольник, можно видеть, что вершины располагаются против часовой стрелки, **рис.** 1.4. Следует отметить, что такой порядок в DirectX выбирается по умолчанию.

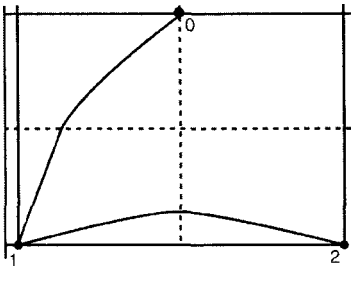


Рис. 1.4. Порядок вершинных доменов

Итак, когда мы знаем, как работает алгоритм исключения невидимой поверхности, стало очевидно, что для нашего простого приложения мы просто не нуждаемся в тех объектах, которые будут исключены. Существует простое состояние рендера (render state), которое управляет режимом отбора и устранения. Добавьте следующую строку к нашему вызову функции SetupCamera:

```
device.RenderState.CullMode = Cull.None;
```

Теперь, когда приложение запущено, все должно работать так, как мы ожидали. Мы имеем плавно вращающийся неисчезающий треугольник. Впервые мы получили реальное трехмерное приложение. Прежде чем продолжить, попробуем слегка изменить размеры окна. Обратите внимание на то, как при этом ведет себя треугольник, его положение и направление вращения не изменились.

Автоматический сброс устройства во время изменения размеров окна

Любой, кто когда-либо писал приложение Direct3D на языке C++ или DirectX для Visual Basic, знает, что при изменении окна вашего приложения устройство должно сброситься или перезапуститься, иначе приложение Direct3D продолжит рендеринг используемой сцены с тем же самым пространственным разрешением, что и раньше, и в результате изображение будет скопировано (и соответственно вытянуто, чтобы заполнить окно) в измененное окно.

Управляемый DirectX способен контролировать процедуру изменения форм окон и автоматически перезапускать устройство при изменении окна. Для этого существует опция, называемая DeviceResizing (изменение окна устройства), формирующая код или процедуру автоматического сброса устройства. Устанавливая данную опцию и определяя значение «true» для отмены (Cancel) класса обработчика событий EventArgs, вы можете сбросить этот параметр в момент изменения окна. Добавьте следующую функцию к вашему примеру:

```
private void CancelResize(object sender, CancelEventArgs e)
1
   e. Cancel = true;
}
```

Как можно видеть, эта функция подтверждает отмену действия. Далее необходимо добавить этот обработчик событий к нашему устройству. Добавьте следующую строку в конце метода создания устройства:

```
device.DeviceResizing += new CancelEventHandler(this.CancelResize);
```

Теперь запустите приложение еще раз и после того, как оно начнет прокручиваться, максимизируйте окно. Картина, которую вы видите далеко не идеальная, треугольник находится в том же самом положении, но теперь его грани имеют неровные края. Необходимо удалить последние

две секции добавленного нами кода. Заданный по умолчанию режим Управляемого DirectX отрабатывает процедуру изменения размеров устройства, и мы можем воспользоваться преимуществом этого режима.

Создание освещения

Теперь, когда мы создали вращающийся треугольник, мы можем добавить освещение. Мы уже кратко упомянули об освещении ранее, когда наш треугольник стал полностью черным после того, как мы использовали непреобразованные треугольники. Фактически в тот момент мы полностью выключили освещение в нашем сценарии. Первое, что мы должны сделать, это — включить освещение, для этого необходимо установить значение «true» в операторе освещения:

```
device.RenderState.Lighting = true;
```

Вы могли бы просто удалить эту строку, поскольку для устройства по умолчанию установлено значение «lighting on» — освещение включено, но оставим это для наглядности. Запуская приложение теперь, легко заметить, что мы опять вернулись к черному вращающемуся треугольнику. Нам необходимо вначале определить свет и уже потом включить его. Обратите внимание, что класс устройства имеет прикрепленный «световой» массив, каждый член которого определяет различные свойства света. Если мы хотим «настроить» первоначальный свет в нашей сцене и включить его, следует добавить следующий код в наш метод OnPaint сразу после кода определения нашего треугольника:

```
device.Lights[0].Type = LightType.Point;
device.Lights[0].Position = new Vector3();
ievice.Lights[0].Diffuse = System.Drawing.Color.White;
device.Lights[0] .Attenuation() = 0.2f;
device.Lights[0].Range = 10000.Of;
device.Lights[0].Commit));
device.Lights[0].Enabled = true;
```

Вначале определяется, какой тип освещения мы хотим отобразить. Мы выбрали точечный источник света, который излучает во всех направлениях одинаково, подобно свече или спирали лампы накаливания. Существует также направленное или параллельное освещение. Например, находясь на Земле, можно думать о Солнце, как о направленном источнике света, если учесть расстояние от Солнца до Земли (хотя в действительности. Солнце скорее является точечным источником, испускающим расходящиеся в пространстве световые волны). Направленный свет характеглвчется только направлением и цветом и исключает все другие свето-

вые коэффициенты, как-то: затухание и диапазон, и является наиболее предпочтительным в вычислительном аспекте его использования. Можно отметить еще один тип светового источника — световое пятно (тип источника, излучающего конический пучок световых лучей), который, как следует из его названия, используется для получения светового пятна на объекте подобно тому, что вы наверняка часто видели на концерте при освещении сцены. Учитывая большое количество коэффициентов в таком варианте освещения (местоположение, направление, конический угол и так далее), такое освещение более громоздко при использовании его в вычислительных операциях.

Продолжим. Далее мы хотим установить положение нашего точечного источника освещения. Так как координаты центра нашего треугольника составляют (0, 0, 0), мы можем там же расположить и наш источник света. Это можно сделать с помощью конструктора параметров parameterless класса Vector3. Мы устанавливаем диффузный компонент осветителя diffuse для получения рассеянного белого света, так чтобы поверхность нормально освещалась. Далее мы задаем параметр затухания attenuation, определяющий изменение интенсивности света при изменении расстояния. Диапазон освещения range — это максимальное расстояние, в пределах которого распространяется свет. В данном случае установленный диапазон значительно превышает наши потребности. В программном пакете DirectX SDK (включенном в CD диск) содержится дополнительная информация о математических нюансах проблем освещения.

Итак, мы применили свет к нашему устройству и включили его. Если посмотреть на параметры освещения, можно обратить внимание, что один из них является логической переменной и называется Deferred (Задержанный). По умолчанию его значением является «false», и поэтому мы использовали параметр Comit, чтобы запустить эту опцию. Установка указанного параметра в значение «true» позволяет не вызывать оператор Comit. Необходимо всегда проверять, что ваше освещение установлено и включено, это позволит предвидеть результаты при выполнении приложения.

Если вы запустите ваше приложение еще раз, то обратите внимание, что даже после определения и установки освещения треугольник все еще остается черным. Причина этого кроется в том, что приложение Direct3D не может освещать объект, не имея нормали к каждой его поверхности. Нормаль представляет собой вектор, перпендикулярный передней стороне объекта, **рис.** 1.5.

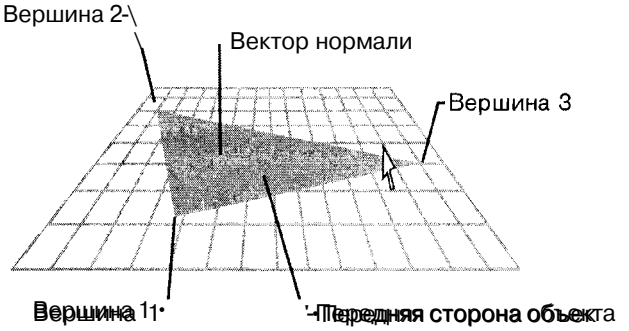


Рис. 1.5. Нормаль к плоскости вершин

Зная это, добавим нормаль к нашему треугольнику. Замените код создания треугольника на следующий:

```
CustomVertex.PositionNormalColored[]  verts=new
CustomVertex.PositionNormalColored[3];
verts[0].SetPosition(new Vector3(0.0f,  l.Of,  l.Of));
verts[0].SetNormal(new Vector3(O.Of,  O.Of,  -l.Of));
verts[0].Color  =  System.Drawing.Color.White.ToArgbf);
verts [l].SetPosition(new Vector3(-1.0f,  -l.Of,  l.Of));
verts [1]. Set-Normal (new Vector3(0.0f, O.Of, -l.Of));
verts[1].Color  =  System.Drawing.Color.  White.ToArgb();
verts[2].SetPosition(new Vector3(1.0f,  -l.Of,  l.Of));
verts[2].SetNormal(new Vector3(0.0f,  O.Of,  -l.Of));
verts[2] .Color  =  System.Drawing.Color.  White.ToArgb();
```

Кроме того, нам также следует изменить сам формат вершин, чтобы согласовать наши новые данные:

```
device.VertexFormat  =  CustomVertex.PositionNormalColored.Format;
```

Единственным значительным изменением между этим набором данных и предыдущим является добавление нормали и того факта, что цвет каждой вершины определяется как белый. Итак, мы определили вектор нормали для каждой вершины, который направлен перпендикулярно передней поверхности. Поскольку положение наших вершин относительно оси Z не изменилось (l.Of), при изменении координат в плоскости X и Y, перпендикулярный вектор попал бы в область отрицательных значений оси Z. Запуская теперь наше приложение, можно увидеть, что треу-

гольник перевернут и освещен. Попробуйте изменить цвет диффузного компонента, чтобы увидеть, как освещение воздействует на сцену. Обратите внимание на то, что, если вы устанавливаете красный цвет диффузного компонента, треугольник окажется освещенным красным светом. Поэксперементируйте со значением этого компонента, чтобы получить различные цветовые варианты.

Следует помнить одну вещь: свет рассчитывается по отношению к вершине, поэтому при использовании полигонов с малым количеством вершин (например, отдельного треугольника), освещение будет менее реалистичным. В последующих главах мы обсудим более совершенные методы освещения, такие как освещение пиксела. Такое освещение ото-, сражает гораздо более реалистичную картину.

Состояния устройства и преобразования

Два элемента, которые мы использовали в нашем коде, до сих пор не были применены к нашему устройству для выполнения каких-либо преобразований. Имеются три различных варианта состояния устройства: состояние отображения (оно же состояние рендера), состояние сэмплера и состояние текстуры. Последние два состояния, использующиеся для текстурирования, мы рассмотрим достаточно скоро. Класс состояния рендера может изменяться в течение того времени, пока Direct3D производит растеризацию сцены. Существует много различных изменяемых параметров и методов этого класса, включая освещение и устранение невидимых поверхностей, которые мы уже использовали в нашем приложении. Другими опциями, которые вы можете устанавливать в пределах состояния рендера, являются fill mode (покадровый режим с сеткой) и различные параметры режима тумана или дымки. Мы обсудим большинство этих опций и процедур в последующих главах.

Как уже упоминалось раньше, существует матрица преобразования одной системы координат в другую. В устройстве используются три главных преобразования: общее (мировая матрица), преобразование вида и преобразование проекции. Однако, есть и другой тип преобразования, который можно дополнительно использовать для изменения состояния текстур, поддерживающий до 255 мировых матриц.

Свопинг-связки и рендеры

Существуют несколько неявных функций и процедур обработки в устройстве, которые определяют, где и как нарисовать объект. Каждое устройство имеет неявные цепочки подкачки или свопинг-связки, использующиеся для рендеринга объектов.

Свопинг-связка представляет собой ряд буферов, управляющих рендерингом. Есть вторичный буфер, определяющий, где выполнить рисование в пределах этой последовательности. Когда свопинг-связка, созданная с помощью метода SwapEffect.Flip готова, данные вторичных буферов передаются на первичный буфер, с которого будет считывать данные ваша графическая плата. В то же самое время третичный буфер становится новым вторичным буфером, а предыдущий первичный буфер «перемещается» в неиспользованный третичный буфер, **рис.1.6.**

Операция перемещения flip изменяет местоположение данных, считываемых видеокартой, и перекачивает старые данные назад в текущее положение вторичного буфера. Для DirectX 9 этот термин используется в общем смысле для сообщения, что вторичный буфер модифицируется как дисплей. В оконном режиме операции перемещения представляют собой копирование данных, подразумевая, что наше устройство не управляет всем дисплеем, а только его частью. В конечном счете, результат остается тем же самым. В полноэкранном режиме, используя SwapEffect.Flip осуществляется фактическое перемещение. Некоторые драйверы также осуществляют операции SwapEffect.Discard или SwapEffect.Copy перемещения в полноэкранном режиме.

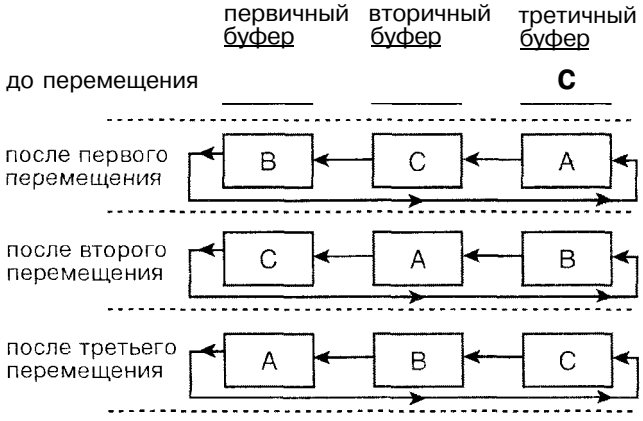


Рис. 1.6. Связи вторичных буферов во время операций перемещения

Если вы создали свопинг-связку, используя SwapEffect.Copy или SwapEffect.Flip, это гарантирует, что любое существующее обращение не будет воздействовать на вторичный буфер свопинг-связки. Приложение runtime воспримет это, создавая при необходимости дополнительные скрытые буферы. Чтобы избежать этого, рекомендуется использовать операцию SwapEffect.Discard. Этот режим позволяет драйверу определять наиболее эффективный способ представления вторичного бу-

фера. Стоит отметить, что при использовании SwapEffectDiscard необходимо убедиться в полной очистке вторичного буфера перед запуском новых операций рисования. Во время отладки приложение runtime заполнит вторичный буфер случайными данными, если предварительно не будет вызвана функция очистки.

ИСПОЛЬЗОВАНИЕ МУЛЬТИВЫБОРОК (МУЛЬТИ-СЭМПЛИНГА) В ВАШЕМ УСТРОЙСТВЕ

Если вы планируете использовать при рисовании сглаживание поверхностей (или мульти-сэмплинг), вам необходимо использовать операцию SwapEffect.Discard. Попытка использовать любую другую операцию будет неудачной.

Вторичный буфер свопинг-связки может использоваться также в качестве рендера. Когда устройство создано, можно использовать созданную свопинг-связку, к вторичному буферу которой привязана поверхность рендеринга устройства. Объект рендеринга представляет собой поверхность, которая будет поддерживать вывод производимых операций рисования. Если вы создаете множественные свопинг-связки обработки различных операций рендеринга, необходимо убедиться, что вы заблаговременно переопределили процедуру рендеринга. Мы подробнее обсудим это в следующих главах.

Краткие выводы

Нарисовав треугольник в нашей сцене, а затем вызвав соответствующую процедуру, мы определили и задали вращение треугольнику. При этом, мы выполнили следующие действия.

- Создали Direct3D устройство, прикрепленное к нашей винформе.
- Нарисовали отдельный цветной треугольник в нашем окне.
- Задали вращение треугольнику.
- Осуществили простое освещение нашей сцены.

Результат выполнения описанных процедур представлен на **рис.1.7.**

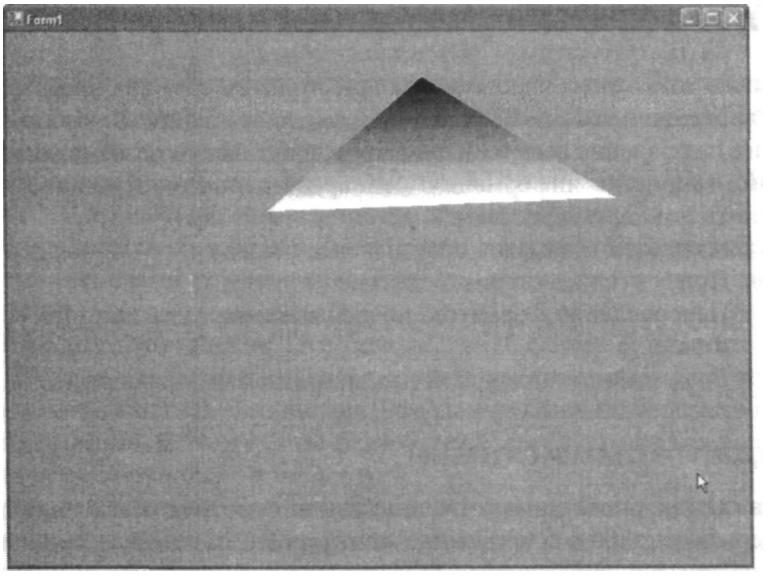


Рис. 1.7. Вращающийся трехмерный треугольник

В следующей главе мы рассмотрим различные опции, которые наше устройство может поддерживать, а также создание других устройств.

Глава 2. Выбор подходящего устройства

Число возможных перестановок при создании того или иного устройства достаточно велико. Имея сегодня на рынке обширный выбор графических плат, знание всех их параметров практически невозможно. Необходимо запросить информацию непосредственно в устройстве, чтобы выяснить его характеристики.

В рамках этой главы, мы обсудим следующее.

- Поиск всех адаптеров в системе.
- Перечисление форматов, поддерживаемых для каждого устройства.
- Возможности и свойства рассматриваемых устройств.

Адаптеры системы

На сегодняшний момент существует множество систем, способных поддерживать работу современных мониторов. Тем не менее, не так много совершенных и эффективных графических карт, способных поддерживать мультимониторный режим «mainstream», использование которых позволяет более широко реализовать возможности отображения графической и видео информации. Самые последние платы ATI, nVidia, и Matrox имеют сдвоенные выходы, эти карты позволяют поддерживать работу нескольких мониторов.

Устройства, созданные в приложении Direct3D, завязаны на используемый адаптер. В данном случае мы можем подразумевать адаптер как отдельное аппаратное устройство компьютерной графики, соединяемое с монитором. Современная карта ATI Radeon 9700 является отдельным физическим адаптером, имеет два выхода для монитора (DVI и VGA), и таким образом, распознается приложением в Direct3D как два адаптера.

Иногда при работе с приложениями необходимо узнать, сколько устройств и какие из них находятся на данный момент в системе и принимают участие в выполнении игры.

В сборках Direct3D существует статический класс, именуемый класс «Manager», который можно использовать для идентификации адаптеров и имеющихся в системе устройств. Самая первая характеристика в классе Manager — список адаптеров в системе. Этот параметр может работать по-разному. Например, он сохраняет значение счетчика «count», сообщающего о количестве адаптеров в системе. Каждый адаптер может быть индексирован непосредственно (например, Manager.Adapters[0]), или это может быть сквозная нумерация адаптеров в вашей системе.

Чтобы продемонстрировать эти характеристики, напишем простое приложение, которое отобразит на экране разветвленную схему, включающую список текущих адаптеров в вашей системе и поддерживаемых

ими режимов отображения. Загрузите Visual Studio, и выполните следующее:

1. Создайте новый проект Windows Form Project для C#, назвав его как угодно, например, для типового кода — «Enumeration» — «Перечисление».

2. Добавьте ссылку на сборки Microsoft.DirectX.Direct3D и Microsoft.DirectX, а также директивы «using» для этих сборок.

3. В представлении проекта для созданной формы windows, добавьте к вашему приложению управление TreeView, находящееся в панели инструментов.

4. Выберите управление TreeView на вашей форме, и нажмите клавишу F4 (или щелкните правой кнопкой мыши, и выберите пункт Properties — Свойства). В свойствах TreeView установите параметр Dock в значение «Fill». В этом случае окно будет всегда заполняться полностью, даже если оно было изменено.

Теперь необходимо добавить функцию, которая просканирует все адаптеры в системе и выдаст некоторую информацию относительно поддерживаемых режимов каждого из них. Добавьте следующие строки где-нибудь в вашем классе:

```
  / <summary>
 / We will fill our tree view here
  / </summary>
public void LoadGraphics()

   foreach(Adapter-Information ai in Manager.Adapters)

      TreeNode root = new TreeNode(ai.Information.Description);
      TreeNode driverlnfo = new TreeNode(string.Format
          ("Driver information: {0} - {1}",
         ai.Information.DriverName,
         ai.Information.DriverVersion) );
      // Add the driver information to the root node
      root.Nodes.Add(driverlnfo);
      // Get each display mode supported
      TreeNode displayMode = new TreeNode(string.Format
          ("Current Display Mode: (0}x{i}x{2}\
         ai.CurrentDisplayMode.Width,
         ai.CurrentDisplayMode.Height,
         ai.CurrentDispIayMode.Format));
      foreach(DisplayMode dm in ai.SupportedDisplayModes)

         TreeNode supportedNode = new TreeNode(string.Format
             ("Supported: (0} x (1 }x {2}",.
            dm.Width, dm.Height, dm.Format));
```

```
        displayMode.Nodes.Add(supportedNode);
    }
    // Add the display modes
    root.Nodes.Add(displayMode);
    // Add the root node
    treeViewl.Nodes.Add(root);
```

Управлять выполнением этого алгоритма довольно просто. Чтобы узнать, что происходит, процедуру выполнения можно всегда прервать. Сначала идет поиск всех адаптеров в системе. Механизм «Foreach», используемый в C#, делает этот программный цикл на удивление простым. Для каждого адаптера в системе цикл выполнится один раз и заполнит структуру Adapterlnformation для данного адаптера.

Структура Adapterlnformation имеет несколько значений:

```
public struct Adapterlnformation
(
int Adapter;
DisplayMode CurrentDisplayMode;
AdapterDetails Information;
AdapterDetails GetWhqllnformationO;
DisplayModeEnumerator SupportedDisplayModes;
}
```

Значение Adapter относится к номеру адаптеру, который вы используете при создании устройства. Порядковые номера адаптера отсчитываются от нулевого индекса с номером ordinals, равным числу адаптеров в вашей системе. Два значения параметра AdapterDetails возвращают идентичные результаты, за одним лишь исключением. В отличие от GetWhqlrnformation, в функции Information не возвращается подробная информация о WHQL (Лаборатории по сертификации аппаратных средств для работы в среде Windows). Извлечение этой информации может быть весьма долгим, поэтому целесообразно отделить данную операцию.

Структура AdapterDetails содержит более подробную информацию непосредственно об адаптере, включая его описание и информацию о драйвере. Приложения могут использовать эту структуру для некоторых типов аппаратных средств, хотя это не всегда является необходимым.

Последние два члена этой структуры возвращают параметры структур DisplayMode, которые могут использоваться для определения различных режимов работы дисплея, включая ширину и высоту визуального

отображения, также как частоту обновления и используемый формат. Значение CurrentDisplayMode возвращает информацию о режиме визуального отображения в используемом адаптере, в то время как SupportedDisplayModes возвращает список всех режимов визуального отображения, поддерживаемых данным адаптером.

Итак, мы используем описание устройства, обнаруженного системой, для создания корневой папки в нашей разветвленной схеме. Затем мы добавляем папку или строку, которая будет включать имя драйвера и номер версии (полученные из той же системы) в качестве ответвлений или в качестве второго уровня этой корневой папки, и добавляем второй уровень схемы, включающий текущие параметры режима визуального отображения адаптера.

Затем мы добавляем в разветвленную схему дочерние узлы или дочерние папки с данными о поддерживаемом формате и режиме этого адаптера. Далее необходимо обеспечить вызов этой функции при запуске приложения, после чего мы готовы запускать приложение. Измените основную функцию следующим образом:

```
using (Forml frm = new Forml())
{
   frm.LoadGraphics lb-
   Application.Run(frm);
}
```

При выполнении данного приложения на экране появится окно, соответствующее рис.2.1.

Как видно из рисунка, этот список содержит список режимов и форматов, поддерживаемых устройством. При заполнении структуры параметров форматы могут использоваться как форматы вторичного буфера. Перечисляемые форматы описываются с помощью буквенно-цифрового кода. Символ представляет тип данных; номер — число битов поддерживаемых данных. Некоторые из символов могут быть следующие:

A — alpha, альфа
X — unused, неиспользуемый
R — red, красный
G — green, зеленый
B — blue, синий
L — luminance, яркость
P — palette, палитра

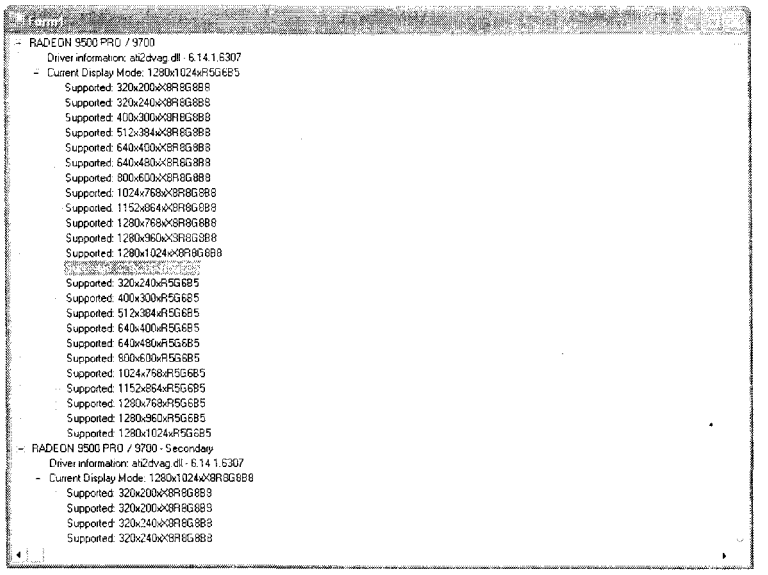


Рис. 2.1. Разветвленная схема режимов адаптера

Полная сумма всех битов в формате определяет полный размер формата. Например, формат X8R8G8B8, **рис.2.1,** означает 32-разрядный формат с 8 битами, используемыми для каждого цвета (красный, зеленый и синий) и с неиспользованными 8 битами.

ФОРМАТЫ ДЛЯ ВТОРИЧНЫХ БУФЕРОВ И ДИСПЛЕЕВ

В большом списке форматов, есть несколько подходящих для использования в качестве вторичного буфера или формата дисплея. Некоторые из них:

A2R10G10B10	A1R5G5B5
A8R8G8B8	X1R5G5B5
X8R8G8B8	R5G6B5

Форматы дисплея могут быть теми же, что и форматы вторичных буферов, за исключением тех, которые содержат символьный компонент. Единственный формат, который может использоваться с символьной компонентой для дисплея, —- это формат A2R10G10B10, но только в полноэкранный режиме.

Обратите внимание, что, поскольку эти форматы поддерживаются приложением Direct3D в качестве форматов вторичных буферов, непосредственно адаптер может не поддерживать их. Для нашего примера, единственными поддерживаемыми форматами, которые мы получили, были X8R8G8B8 и R5G6B5.

Итак, мы собрали достаточно информации, чтобы определить порядковый адаптер для устройства, которое хотим создать, а также вторичный буферный формат, который мы хотим поддерживать.

Поддержка аппаратного устройства

Класс Manager содержит множество функций, которые могут использоваться для определения возможности поддержки имеющегося адаптера. Если вы хотите определить, действительно ли ваше устройство поддерживает определенный формат, и не хотите перебирать все возможные адаптеры и форматы, вы можете использовать для этого класс Manager. Используйте следующую функцию:

```
public static System.Boolean CheckDeviceType ( System.Int32 adapter ,
    Microsoft.DirectX.Direct3D.DeviceType checkType ,
    Microsoft.DirectX.Direct3D.Format displayFormat ,
    Microsoft.DirectX.Direct3D.Format baciBufferformat ,
    System.Boolean windowed , System.Int32 result )
```

Данная функция позволяет быстро определить, поддерживает ли ваше устройство используемый тип формата. Первый параметр — *adapter,* тестируемый адаптер. Второй — *checktype,* тип проверяемого устройства (как правило, это DeviceType.Hardware). Окончательно определяются отображаемый формат *displayFormat* и формат вторичного буфера *Ьаск-BufferFormat,* а затем, режим работы (оконный (windowed) или полноэкранный). Последний параметр — дополнительный, при использовании он возвратит целочисленный код (HRESULT в COM) функции. Данный алгоритм возвращает значение «true», если данный тип устройства поддерживается, в противном случае возвращает значение «false».

ПРЕОБРАЗОВАНИЕ ОПРЕДЕЛЯЕМОГО ФОРМАТА

Важно обратить внимание, что в оконном режиме формат вторичного буфера не должен соответствовать формату режима визуального отображения, если аппаратные средства могут поддерживать соответствующее цветовое преобразование. Метод CheckDeviceType возвращает соответствующие результаты, независимо от того, доступен данный формат или нет, также для определения этого напрямую возможно использование метода CheckDeviceFormatConversion класса Manager. В оконном режиме вы можете использовать формат Format.Unknown.

Было бы весьма полезно знать заранее те типы форматов, которые вы будете использовать. Это освобождает от необходимости перебирать все возможные форматы и устройства.

Проверка возможностей устройства

Различные возможности аппаратных средств описываются термином «capability» — возможность. В приложении Direct3D runtime имеется опция, которая перечисляет все реализуемые возможности используемого устройства. Как только устройство создано, вы можете использовать опцию устройства capability для определения поддерживаемых возможностей. Если устройство еще не создано, вы так же можете узнать возможные поддерживаемые характеристики этого устройства с помощью определенного метода класса Manager.

Теперь можно добавить к нашему приложению код, который выдаст описание свойств и возможностей каждого адаптера в нашей системе. Однако, мы не можем добавить список возможностей к имеющейся разветвленной схеме драйверов и параметров из-за их большого числа. Самый простой способ показать эти данные состоит в использовании текстового поля.

Теперь вернемся в процедуру view нашей формы и попробуем переключать параметры разветвленной схемы представления от позиции «Fill» до позиции «Left». Уменьшите размер окна вполовину. Теперь добавьте текстовое поле к окну, установите параметр Dock в положение «Fill». Также убедитесь, что строка «Multiline» установлена в положение «true», и параметр прокрутки «Scrollbars» установлен в положение «Both» для текстового поля.

Далее нам следует добавить обработчик событий к приложению так, чтобы он создавал текстовое поле с параметрами обнаруженного адаптера. Для этого необходимо привязать опцию «AfterSelect» к событию обнаружения адаптера (предполагается, что вы уже знаете, как отслеживать эти события). Используйте следующие команды:

```
private void treeViewl_AfterSelect(object
sender,System.Windows.Forms.TreeViewEventArgs e)
(
   if (e.Node.Parent == null) // root node
   (
      // Get the device caps for this node
```

```
textBoxl.Text = e.Node.Text + " Capabilities: \r\n\r\n" +
Manager.GetDeviceCaps
    (e.Node.Index, DeviceType.Hardware).ToString().Replace("\n", "\r\n");
```

Как видите, это достаточно просто. Для корневых папок, которые отображают названия адаптеров после того, как они выбраны, мы вызываем функцию Manager.GetDeviceCaps для соответствующего номера адаптера. Параметр ToString этой строки возвращает чрезвычайно большой список всех возможностей данного устройства. Выполнение данного приложения приведет к появлению на экране более подробной разветвленной схемы, см. пример на рис.2.2.

ИСПОЛЬЗОВАНИЕ ОПЦИИ «CAPABILITY» ДЛЯ НЕПРОГРАММИРУЕМОГО КОНВЕЙЕРА

Многие из возможностей, перечисленных в структуре «capability» связаны непосредственно с непрограммируемым конвейером (например, значение MaxActiveLights). Если вы используете программируемый конвейер, многие из возможностей не будут использованы в вашей сцене. Позднее мы обсудим различия между непрограммируемым и программируемым конвейерами.

Итак, теперь у нас имеется полный список возможностей, которые могут поддерживаться устройством. Например, мы можем найти информацию, поддерживает ли ваше устройство режим вуализации (fogging), или узнать максимальное число одновременно выполняемых рендеров. Структура «Capability» разбита главным образом на две группы. Первая группа — логические значения, которые определяют, поддерживается ли данная аппаратная возможность или нет. Например, SupportsAlphaCompraге принимает значение «true», если устройство поддерживает текстовые символы. Другая группа класса «Capability» возвращает «фактические значения, например значение MaxActiveLights, которое определяет максимальное число активных подсветок, реализуемых в сцене. Далее во мере необходимости мы рассмотрим и опишем отдельные возможности устройства.

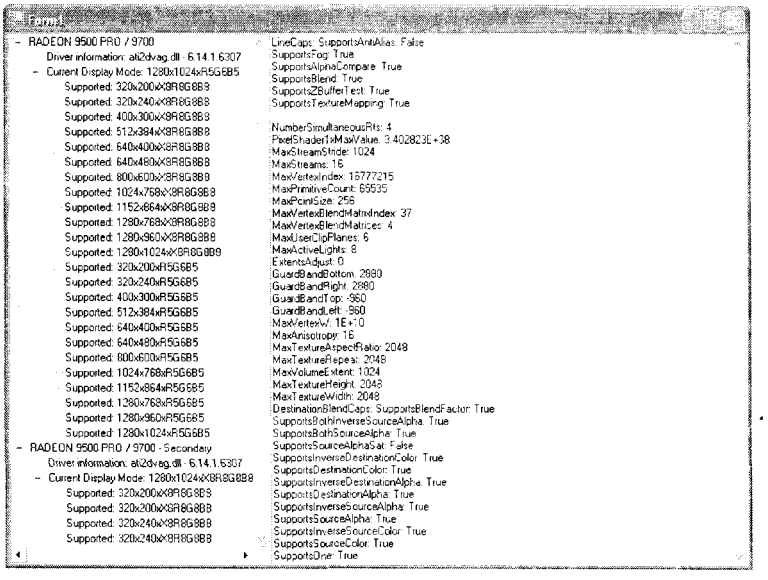


РИС. 2.2. Возможности и режимы обнаруженных устройств

Краткие выводы

В этой главе мы рассмотрели выбор подходящего графического устройства для работы с нашим приложением. В главе были охвачены следующие пункты.

Поиск и обнаружение всех имеющихся в системе адаптеров.

- Определение форматов, поддерживаемых нашим устройством.
- Определение и описание возможностей имеющихся устройств.

В следующей главе мы продолжим совершенствовать навыки построения изображений, включая новые возможности текстурирования для получения более реалистичных сцен.

Глава 3. Введение в технологию рендеринга

То, что мы подразумевали под рендерингом и использовали до сих пор, было не слишком эффективно. Новые значения вершин перераспределялись каждый раз, когда отображалась сцена, и данные сохранялись в системной памяти. При этом необходимо было при каждом отображении копировать данные о вершинах из системной памяти в память графической карты, это требует большего времени выполнения и является неэффективным. Использование современных графических плат, имеющих встроенную собственную память, позволяет значительно расширить возможности отображения изображений, сохраняя данные о вершинах непосредственно в видеопамяти платы.

Использование вершинных буферов

Приложение Direct3D имеет необходимый для этого механизм — вершинные буферы. Такой буфер, как следует из его названия, является памятью для хранения данных о вершинах. Гибкость вершинных буферов делает их идеальными в плане геометрических преобразований в используемой сцене. Попробуем усовершенствовать наш треугольник, описанный в первой главе, применяя вершинные буферы.

Создание вершинного буфера достаточно несложно. Существует три конструктора, которые можно использовать для этого. Рассмотрим по отдельности каждый из них.

```
public VertexBuffer ( Microsoft.DirectX.Direct3D.Device device ,
   System.Int32
   sizeOfBufferInByt.es , Microsoft.DirectX.Direct3D.Usage usage ,
   Microsoft.DirectX.Direct3D.VertexFormats vertexFormat ,
   Microsoft.DirectX.Direct3D.Pool pool )
public VertexBuffer ( System.Type typeVertexType , System.Int32 numVerts ,
   Microsoft.DirectX.Direct3D.Device device ,
   Microsoft.DirectX.Direct3D.Usage usage ,
   Microsoft.DirectX.Direct3D.VertexFormats vertexFormat ,
   Microsoft.DirectX.Direct3D.Pool pool )
```

Опишем каждый параметр.

- *Device* — устройство, которое используется в Direct3D для создания вершинного буфера. Данный буфер будет поддерживаться только этим устройством.
- *SizeOfBufferlnBytes* — задаваемый размер вершинного буфера в байтах. Если вы используете в конструкторе этот параметр, буфер способен поддерживать любой тип вершин.

TypeVertexType — задается, если вы хотите, чтобы буфер включал только один тип вершин. Либо это может быть тип одной из встроенных структур вершин в классе CustomVertex, либо тип вершин, определяемый пользователем. Данное значение не может быть пустым.

NumVerts — максимальное количество вершин, которые вы хотите сохранить в буфере. Это значение должно быть положительным и ненулевым.

- *Usage* — определяет, как этот вершинный буфер может использоваться. При создании буфера могут использоваться все значения данного параметра, за исключением следующих.

  DoNotClip — используется для того, чтобы исключить режим clipping (отсечения) для этого буфера. Используя этот флажок, необходимо при использовании рендеринга из вершинного буфера установить параметр clipping состояния рендера в значение «false».

  Dynamic — используется для сообщения приложению, что вершинный буфер требует использования динамической памяти. Если этот флажок не определен, вершинный буфер является статическим. Статические вершинные буферы обычно хранятся в видеопамяти, в то время как динамические буферы хранятся в памяти AGP. Выбор данной опции необходим для драйверов, чтобы определить, где сохранить данные. Более подробно с режимом Dynamic можно ознакомиться в документации на DirectX SDK.

  - Npatches — сообщает, что данный буфер используется для рисования патчей N-Patches.

    Points — сообщает, что данный буфер используется для рисования точек.

    RTPatches — сообщает, что данный буфер используется для рисования примитивов более высокого порядка.

    SoftwareProcessing — сообщает, что обработка вершин должна проводиться программным обеспечением, в противном случае обработка ведется аппаратными средствами.

  - WriteOnly — сообщает, что вершинный буфер не будет считываться в случае, когда нет необходимости считывать данные вершинного буфера. Данная опция позволяет разгрузить видеопамять.

- *VertexFormat* — определяет формат вершин, которые будут сохранены в этом буфере. Вы можете выбирать опцию VertexFormat.None, если планируете зарезервировать этот буфер.
- *Pool* — определяет пул памяти, куда вы хотите поместить вершинный буфер. Вам следует выбрать одно из следующих размещений пула памяти.

- Default — вершинный буфер размещен в памяти, объединяет большинство размещенных в ней данных. Данные располагаются либо в видеопамяти, либо в памяти AGP, в зависимости от параметра использования. Вершинные буферы, созданные в этом пуле памяти, переопределяются автоматически перед сбросом устройства.
- SystemMemory — данные вершинного буфера помещаются в системную память, где являются недоступными для устройства.
- Scratch — системный пул памяти, не связанный с устройством и не используемый устройством. Удобен при управлении данными, при этом не привязан к специфическому формату устройства.

СОЗДАНИЕ ВЕРШИННЫХ БУФЕРОВ С ПОМОЩЬЮ НЕУПРАВЛЯЕМЫХ COM УКАЗАТЕЛЕЙ

Как и в самом устройстве, в приложении имеется перегрузка для вершинного буфера, которая принимает указатель IntPtr. Это значение используется для передачи указателя интерфейса COM в неуправляемый интерфейс IDirect3DvertexBuffer9. Это удобно, когда вы хотите использовать вершинный буфер, созданный с помощью внешнего (неуправляемого) источника. Любое другое управляемое приложение никогда не будет использовать данный конструктор, и это значение не может быть пустым или нулевым.

Вспоминая цветной треугольник, созданный в главе 1, можно достаточно легко переместить данные треугольника в вершинный буфер. Вначале, сразу после кода создания нашего устройства, объявите переменные вершинного буфера:

```
private Device device = null;
private VertexBuffer vb = null;
```

Вместо того чтобы каждый раз создавать данные при отображении сцены, можно сделать это один раз. Использование вершинного буфера позволяет в значительной степени увеличить производительность операций отображения рисунка. Для записи данных сразу после создания устройства перепишите код создания треугольника:

```
// Create our device
device = new Device(0, DeviceType.Hardware, this,
CreateFlags.SoftwareVertexProcessing
    , presentParams);
```

```
CustomVertex.PositionColored[] verts = new CustomVertex.PositionColored[3];
verts[0].SetPosition(new Vector3(0.Of, I.Of, l.Of));
verts [0] .Color = System.Drawing.Color.Aqua.ToArgb();
verts[l].SetPosition(new Vector3(-1.0f, -l.Of, l.Of));
verts [1] .Color = System.Drawing.Color.Black.ToArgb()
verts[2].SetPosition(new Vector3(l.Of, -l.Of, l.Of));
verts [2] .Color - System.Drawing.Color.Purple.ToArgbO ;
vb = new VertexBuffer(typeof(CustomVertex.PositionCoiored), 3, device,
Usage.Dynamic :
   Usage.WriteOnly, CustomVertex.PositionCoiored.Format, Pool.Default);
vb.SetData(verts, 0, LockFlags.None);
```

Единственные изменения, появившиеся здесь, — две новых строки после кода создания треугольника. Мы сначала создаем вершинный буфер для записи трех значений вершинной структуры, которые мы уже объявили. Мы хотим, чтобы буфер был настроен только на запись, был динамическим и сохранялся только в заданном по умолчанию пуле памяти для увеличения скорости выполнения. Далее необходимо поместить наш список вершин треугольника в вершинный буфер, используя метод SetData. Этот метод принимает любой общий объект (подобно методу DrawUserPrimitives) в качестве первого значения «verts». Второе значение «offset» — величина смещения размещаемых данных. Поскольку мы хотим записать все данные, принимаем здесь смещение, равное нулю. Последний параметр LockFlags.None определяет то, как мы хотим блокировать буфер на то время, пока данные записываются. Различные механизмы блокировки мы опишем несколько позже.

При компилировании данного приложения система выдаст сообщение об ошибке компиляции, поскольку функция DrawUserPrimitives при обращении к OnPaint затребует переменную «verts». Необходимо сообщить Direct3D, что мы хотим рисовать объект именно из нашего вершинного буфера, а не из массива, который мы объявили прежде.

Естественно, такой метод существует. Мы можем вызывать метод SetStreamSource из устройства, чтобы дать возможность Direct3D считывать наш вершинный буфер при рисовании примитивов. Прототипы для двух перегрузок этой функции имеют вид:

```
public void SetStreamSource ( System.Int32 streamNumber ,
   Microsoft.DirectX.Direct3D.VertexBuffer streamData ,
   System.Int32 offsetlnBytes , System.Int32 stride )

public void SetStreamSource ( System.Int32 streamNumber ,
   Microsoft.DirectX.Direct3D.VertexBuffer streamData ,
   System.Int32 offsetlnBytes )
```

Единственным различием между этими двумя перегрузками является то, что каждая содержит дополнительное значение для размера шага потока. Первый параметр *streamNumber* — номер потока, который мы будем использовать для этих данных. В основном мы будем использовать нулевое значение, однако, позже мы обсудим и множественные потоки. Второй параметр — вершинный буфер, который содержит данные нашего источника. Третий параметр *offsetlnBytes* — смещение (в байтах) в вершинном буфере, определяет место в буфере, с которого Direct3D будет рисовать объект. Параметр размера *stride* (для одной перегрузки) определяет размер каждой вершины в буфере. Если вы создали вершинный буфер с использованием типа, нет необходимости применять перегрузку с этим параметром.

Замените обращение к методу рисования drawing следующими строками:

```
device.SetStreamSource(0, vb, 0);
device.DrawPrimitives(PrimitiveType.TriangleList, 0, 1);
```

Как уже отмечалось, мы записываем нулевое значение номера потока данных вершинного буфера, и поскольку хотим использовать все данные, выбираем нулевое значение смещения. Обратите внимание, что мы также изменили фактический вызов создания рисунка. Поскольку мы имеем все наши данные в вершинном буфере, нет необходимости вызывать метод DrawUserPrimitives (данная функция предназначена для рисования объектов, данные о которых непосредственно определяются пользователем). Более универсальная функция DrawPrimitives рисует примитивы из нашего потокового источника. Метод DrawPrimitives имеет три параметра, первый — тип примитива, который мы уже обсудили. Второй параметр — начальная вершина в потоке. Последний параметр — число примитивов, которые мы будем рисовать.

Даже эта простая демонстрация отображения треугольника с использованием вершинного буфера показывает увеличение быстродействия приблизительно на 10%, в основном за счет скорости передачи кадров. Однако, существует определенная проблема, связанная с этим приложением, появляющаяся при попытке изменить размеры окна. В этом случае треугольник просто исчезает.

Есть несколько причин, вызывающих такое поведение, о двух из них уже кратко упоминалось. В предыдущей главе было сказано, что при изменении размеров окна устройство автоматически сбрасывалось. Это означает, что когда после сброса в пуле памяти (например, вершинном буфере) создается ресурс, этот пул устанавливается автоматически при сбрасывании устройства. Таким образом, во время изменения окна уст-

ройство сбрасывается, и устанавливается вершинный буфер. Одной из изящных особенностей Управляемого DirectX является то, что он автоматически воссоздает вершинный буфер после того, как устройство сбрасывается, при этом в буфере не будет никаких данных, и, следовательно, ничего не отобразится на экране.

Вершинный буфер имеет обработчик события «created», который отслеживает момент создания буфера и готовности его к заполнению. Для использования этого обработчика событий необходимо переписать наше приложение, добавив следующий код:

```
private void OnVertexBufferCreate(object sender, EventArgs e)
(
   VertexBuffer buffer = (VertexBuffer)sender;
   CustomVertex.PositionColored[] verts = new
   CustomVertex.PositionColored[3];
   verts[0].SetPosition(new Vector3(0.0f, 1.0f, 1.0f ));
   verts[0].Color = System.Drawing.Color.Aqua.ToArgb();
   verts[1].SetPosition(new Vector3(-1.0f, -1.0f, 1.0f));
   verts[1].Color = System. Drawing.Color.Black.ToArgb();
   verts[2].SetPosition(new Vector3(1.0f, -1.0f, 1.0f));
   verts[2].Color = System.Drawing.Color.Purple.ToArgb();
   buffer.SetData(verts, 0, LockFlags.None);
1
```

Эта функция имеет стандартное описание обработчика событий, включая регистрацию события и список параметров. В нашем случае список параметров не используется, так что это значение можно проигнорировать. Таким образом, вначале создается вершинный буфер, затем отслеживается данное событие, и значение сендера передается назад в вершинный буфер. Далее мы следуем тем же путем, что и в последний раз, когда создавали наши данные и вызывали метод SetData. Мы должны переписать код создания треугольника в методе инициализации InitializeGraphics, используя вместо него следующий две строки:

```
vb.Created += new EventHandler(this.OnVertexBufferCreate);
OnVertexBufferCreate(vb, null);
```

Данная процедура отслеживает описанное событие и вызывает метод OnVertexBufferCreate всякий раз, когда создается вершинный буфер. Так как событие не было отслежено при первоначальном создании объекта, для этого случая необходимо вручную вызвать функцию обработчика событий. Теперь запустите приложение еще раз и попробуйте изменить размеры окна.

ВРЕМЯ РАБОТЫ РЕСУРСА

Все графические ресурсы будут размещаться автоматически, если в течение сброса устройства они сохранены в видеопамяти; однако, после сброса заново пересоздаются только вершинный и индексные буферы. Также важно отметить, что все ресурсы размещаются только тогда, когда опознано само устройство.

Итак, используя вершинные буферы и видеопамять, мы успешно усовершенствовали наше приложение для отображения трехмерного треугольника. Теперь, используя дополнительные рендеры, можно попробовать сделать процесс рисования более захватывающим. Методы и код станут более сложными, но основные принципы сохранятся.

Как вы знаете, вся геометрия в трехмерной сцене составлена из полигонов, в частности из треугольников, даже если вы отображаете куб, поскольку квадрат или прямоугольник можно сложить из двух треугольников, а куб можно создать из шести квадратов. Единственные координаты, которые необходимы для создания куба, — это восемь угловых вершин куба. Давайте изменим наш алгоритм создания геометрии, см. листинг 3.1:

Листинг 3.1. Создание куба или параллелепипеда.

```
CustomVertex.PositionColored[] verts = new
CustomVertex.PositionColored[36];
// Front face
verts[0] = new CustomVertex.PositionColored(-1.0f, 1.Of, 1.Of,
Color.Red.ToArgb()) ;
verts[1] = new CustomVertex.PositionColored(-1.0f, -1.Of, 1.Of,
Color.Red.ToArgb());
verts[2] = new CustomVertex.PositionColored(1.0f, 1.Of, 1.Of,
Color.Red.ToArgb());
verts[3] = new CustomVertex.PositionColored(-1.0f, -1.Of, 1.Of,
Color.Red.ToArgb());
verts[4] = new CustomVertex.PositionColored(1.0f, -1.Of, 1.Of,
Color.Red.ToArgb());
verts[5] = new CustomVertex.PositionColored(1.Of, 1.Of, 1.Of,
Color.Red.ToArgb());
// Back face (remember this is facing *away* from the camera, so vertices
   should be
clockwise order)
verts[6] = new CustomVertex.PositionColored(-1.0f, 1.Of, -1.Of,
Color. Blue.ToArgb());
verts[7] = new CustomVertex.PositionColored(1.Of, 1.Of, -1.Of,
Color.Blue.ToArgb());
```

```
verts[8] = new CustomVertex.PositionColored(-1.0f, -l.Of, -l.Of,
Color.Blue.ToArgb());
verts[9J = new CustomVertex.PositionCoiored(-1.0f, -l.Of, -l.Of,
Color.Blue.ToArgb());
verts[lO] = new CustomVertex.PositionCoiored(1.0f, l.Of, -l.Of,
Color.Blue.ToArgb(l);
verts[ll] = new CustomVertex.PositionColored(1,Of, -l.Of, -l.Of,
Color.Blue.ToArgb());
// Top face
verts[12] = new CustomVertex.PositionColored(-1.0f, l.Of, l.Of,
Color.Yellow.ToArgb());
verts[13] = new CustomVertex.PositionColored(1.0f, l.Of, -l.Of,
Color.Yellow.ToArgb()) ;
verts[14! = new CustomVertex.PositionColored(-1.0f, l.Of, -l.Of,
Color.Yellow.ToArgb());
verts[15J = new CustomVertex.PositionColored(-1.0f, l.Of, l.Of,
Color.Yellow.ToArgb());
verts[16] = new CustomVertex.PositionColored(1.0f, l.Of, l.Of,
Color.Yellow.ToArgb());
verts[17] = new CustomVertex.PositionCoiored(1.0f, l.Of, -l.Of,
Color.Yellow.ToArgb());
// Bottom face (remember this is facing *away* from the camera, so vertices
   should be
clockwise order)
verts[18] = new CustomVertex.PositionColored(-1.0f, -l.Of, l.Of,
Color.Black.ToArgb());
verts[19] = new CustomVertex.PositionColored(-1.0f, -l.Of, -l.Of,
Color.Black.ToArgb());
verts[20] = new CustomVertex.PositionColored(l.Of, -l.Of, -l.Of,
Color.Black.ToArgb());
verts[21j = new CustomVertex.PositionColored(-1.0f, -l.Of, l.Of,
Color.Black.ToArgb());
verts[22] = new CustomVertex.PositionColored(1.0f, -l.Of, -l.Of,
Color.Black.ToArgb());
verts[23] = new CustomVertex.PositionCoiored(1.0f, -l.Of, l.Of,
Color.Black.ToArgb());
// Left face
verts[24] = new CustomVertex.PositionColored(-1.0f, l.Of, l.Of,
Color.Gray.ToArgb());
verts[25] = new CustomVertex.PositionColored(-1.0f, -l.Of, -l.Of,
Color.Gray.ToArgb());
verts[26] = new CustomVertex.PositionColored(-1.0f, -l.Of, l.Of,
Color.Gray.ToArgb());
verts[27] = new CustomVertex.PositionColored(-1.0f, l.Of, -l.Of,
Color.Gray.ToArgb());
verts[28] = new CustomVertex.PositionColored(-1.0f, -l.Of, -l.Of,
Color.Gray.ToArgb());
```

```
verts[29] = new CustomVertex.PositionColored(-1.0f, 1.Of, 1.Of,
Color.Gray.ToArgbO);
// Right face (remember this is facing *away* fiom the camera, so vertices
   should be
   clockwise order)
verts[30] - new CustomVertex.PositionColored(1.0f, l.Of, l.Of,
Color.Green.ToArgbO);
verts[31] = new CustomVertex.PositionCoiored(1.0f, -l.Of, l.Of,
Color.Green.ToArgbO);
verts[32] = new CustomVertex.PositionColored(1.0f, -l.Of, -l.Of,
Color.Green.ToArgbO);
verts[33] = new CustomVertex.PositionColored(1.0f, l.Of, -l.Of,
Color.Green.ToArgbf));
verts[34] = new CustomVertex.PositionColoredfl.Of, l.Of, l.Of,
Color.Green.ToArgbO);
verts[35] = new CustomVertex.PositionColored(1.0f, -l.Of, -l.Of,
Color.Green.ToArgbO);
buffer.SetData(verts, 0, LockFlags.None);
```

В данном методе мы формируем необходимое количество вершин для создания нашего объекта, в данном случае 36 вершин. При этом вы можете использовать исходник программы, имеющейся на CD диске. Как уже упоминалось, кубический объект будет создан из 12 треугольников, каждый треугольник имеет 3 вершины. Выполнение данного кода аналогично процедуре обращения к функции SetData.

Следует отметить, что мы никогда не изменяли первоначальный размер нашего вершинного буфера. Таким образом, перед выполнением процедуры необходимо внести пару изменений. Перепишите строки следующим образом:

```
vb = new VertexBuffer(typeof(CustomVertex.PositionColored), 36, device,
   Usage.Dynamic
   ! Osage.WriteOnly, CustomVertex.PositionColored.Format, Pool.Default);
device.Transform.World = Matrix.RotationYawPitchRoll(angle /
   (float)Math.PI, angle /
   (float)Math.PI * 2.Of, angle / (float)Math.PI);
device.DrawPrimitives(PrimitiveType.TriangleList, 0, 12);
```

Главное, что мы здесь делаем, — это изменение размера вершинного буфера, который мы создали для отображения всех необходимых для рендеринга данных. Мы также пытаемся изменить и характер вращения, чтобы получить более эффектную картинку. Наконец, мы пытаемся отобразить 12 примитивов более эффективно, чем мы отображали наш один единственный треугольник. Действительно, поскольку наш новый кубический объект сформирован полностью как 3D-объект и является полностью заполненным и описанным, у нас больше нет необходимости моде-

пировать и отображать обратные невидимые стороны треугольников. Мы можем использовать заданный по умолчанию в Direct3D режим отбора (против часовой стрелки). Поэтому необходимо убрать строку, описывающую режим отбора «cullig», из нашего источника. Затем попробуем повторно выполнить приложение.

Мы получили на экране красочный вращающийся куб с разноцветными гранями. При этом мы видим каждую грань без пропаданий.

Теперь, если бы вы хотели отобразить более чем один трехмерный объект в данной сцене, было бы неэффективно создавать несколько вершинных буферов (один буфер для каждого объекта). Существует намного более простой способ делать это.

Будем рисовать фигуру из трех кубических объектов, некоторые грани которых соприкасаются. Поскольку наши текущие параметры камеры настроены на наш первый куб, для охвата всей сцены мы переместим камеру немного назад. Измените параметр Look в функции следующим образом:

```
device.Transform.View = Matrix.LookAtLH(new Vector3(0,0, 18.0f), new
   Vector3(),
   new Vector3(0,1,0));
```

Теперь, переместив назад нашу камеру, мы можем видеть практически всю сцену. Отображаемый объект в нашей сцене будет казаться меньше по размеру, так как камера расположена от него дальше. Чтобы нарисовать еще два кубических объекта, соприкасающихся с первым, мы можем многократно использовать наш имеющийся вершинный буфер, давая команду приложению Direct3D рисовать ту же самую вершину снова. Добавьте следующие строки программы после обращения к DrawPrimitives:

```
device.Transform.World = Matrix.RotationYawPitchRoll(angle /
    (float)Math.PI, angle /
    (float)Math.PI / 2.0f, angle / (float)Math.PI * 4.0f) *
   Matrix.Translation(5.0f, 0.0f, 0.0f);
device.DrawPrimitives(PrimitiveType.TriangleList, 0, 12);
device.Transform.World = Matrix.RotationYawPitchRoll(angle /
    (float)Math.PI, angle /
    (float)Math.PI * 4.0f, angle / (float)Math.PI / 2.0f) *
   Matrix.Translation(-5.0f, 0.0f, 0.0f);
device.DrawPrimitives(PrimitiveType.TriangleList, 0, 12);
```

Приложение Direct3D уже «знает», какой тип вершин мы планируем рисовать, поскольку с помощью параметра VertexFormat мы определили для рисования наш первый куб. Благодаря функции SetStreamSource (пока только для первого куба) приложению Direct3D также «известно», из ка-

изго вершинного буфера извлекаются данные. Что же необходимо знать приложению Direct3D, чтобы начать рисовать второй и третий куб? Единственная информация, которая необходима, это — где и что рисовать.

Для этого будем преобразовывать координаты первого куба в пространстве мировых координат, другими словами, применим матрицу преобразования. Вначале используем вращение, подобно тому, что мы делали, используя функцию SetupCamera (хотя наша функция несколько отличается, поскольку наши кубические объекты вращаются под различными углами). Затем преобразуем мировые координаты, это достаточно новая операция для нас. Мы умножаем матрицу перемещения (Matrix.Translation) на нашу существующую матрицу вращения. Матрица перемещения изменяет координаты вершин объекта от одной точки к другой в мировом пространстве. В итоге, мы хотим переместить второй кубический объект на пять единиц вправо, а третий объект переместить на пять единиц влево от нашего первого куба.

Итак, преобразования координат будут выполняться в следующем порядке: вершина объекта будет сначала вращаться, затем переместится. При другой последовательности действий результат был бы другим (достаточно вспомнить, что умножение матриц слева направо и справа наJСВО дает различные результаты), поэтому важно помнить порядок операций при преобразовании координат вершин.

В составе CD-диска включен исходник программы, которая позволяет рисовать девять кубических объектов быстрее, чем три, которые описаны здесь, рис.3.1.

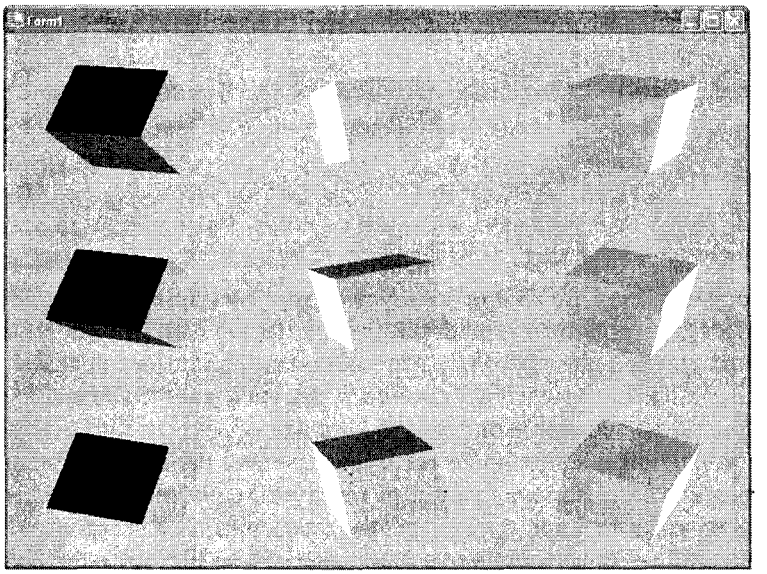


Рис. 3.1. Раскрашенные кубы

Текстурирование объектов

Несмотря на использование рендеринга с цветами и подсветкой, объекты все равно выглядят не так реалистично, как хотелось бы. Термин «текстурирование» («texture») при описании нетрехмерных приложений обычно описывает шероховатость или неровность рисуемого объекта. Текстуры в трехмерной сцене — это, по существу, плоские двухмерные bitmap-рисунки, которые могут использоваться для моделирования текстуры на примитиве. Например, вы могли бы взять точечный рисунок травы, чтобы отобразить реалистичный холм, или, возможно, облака, чтобы отобразить небо. Приложение Direct3D может отображать до восьми текстур одновременно для каждого примитива, но пока давайте разберем одну текстуру на одном примитиве.

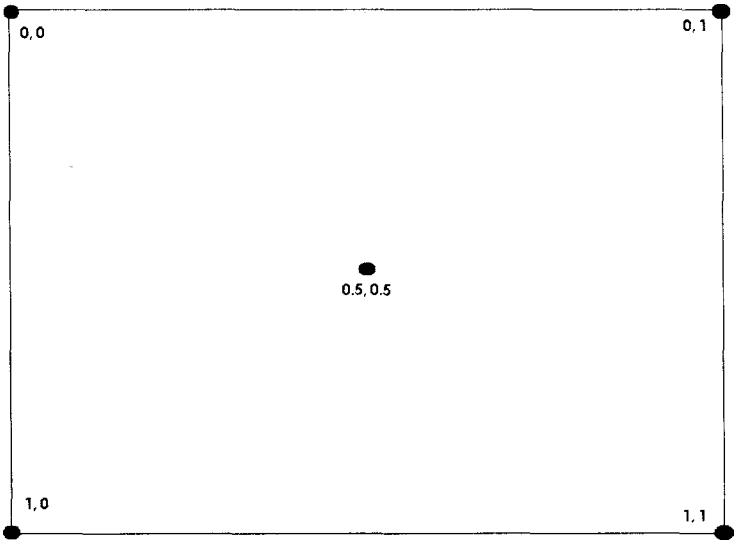


Рис. 3.2. Визуализация текстурных координат

В то время как Direct3D использует структуру bitmap в качестве формата текстуры, любой точечный рисунок, который вы загружаете, может быть использован для текстурирования объекта. Каким же образом можно преобразовать плоский 2D точечный рисунок в нечто такое, что может нарисовать трехмерный объект? Каждый объект, представляемый в сцене, запрашивает координаты текстуры, которые используются для отображения каждого тексела (элемента текстуры) в соответствующем пикселе на экране в течение процедуры растеризации. *Тексел,* от английского слова texel, — сокращенное название элемента текстуры, или соответствующее значение цвета для каждого адреса в текстуре. Под адресом

можно подразумевать номер строки и столбца, называемые «U» и «V», соответственно. Обычно это скалярные значения, диапазон которых составляет от 0,0 до 1,0. Значение 0,0 соответствует верхнему левому углу расположения текстуры, тогда как 1,1 соответствует правому нижнему углу расположения текстуры. Центр текстуры соответствует значению 0,5, 0,5, **рис.3.2.**

Для того чтобы отобразить наши кубические объекты с использованием текстурирования, мы должны изменить вершинный формат нашего объекта ровно также, как и данные, пересылаемые в графическую карту. Мы заменим компонент «color» (цвет) данных наших вершин данными координат текстуры. Для того чтобы иметь наш объект и цветным, и текстурным, мы просто будем использовать текстуру для определения цвета каждого примитива. Перепишите код создания вершины, как показано в листинге 3.2:

Листинг 3.2. Данные текстур куба.

```
CustomV'ertex.PositionTextured[] verts = new
CustomV'ertex.PositionTextured [36];
// Front face
verts[0] = new CustomVertex.PositionTextured(-1.0f, l.Of, l.Of, O.Of,
O.Of);
verts[1] = new CustomVertex.PositionTextured(-1.0f, -l.Of, l.Of, O.Of,
l.Of);
verts[2] = new CustomVertex.PositionTextured(1.0f, l.Of, l.Of, l.Of, O.Of);
verts[3] = new CustomVertex.PositionTextured(-l.Of, -l.Of, l.Of, O.Of,
l.Of);
verts[4] = new CustomVertex.PositionTextured(1.0f, -l.Of, l.Of, l.Of,
l.Of);
verts[5] = new CustomVertex.PositionTextured(1.0f, l.Of, l.Of, l.Of, O.Of);
// Back face (remember this is facing *away* from the camera, so vertices
   should be
   clockwise order)
verts[6] = new CastomVertex.PositionTextured(-1.0f, l.Of, -l.Of, O.Of,
0.Of);
verts[7] = new CustomVertex.PositionTextured(i.Of, l.Of, -l.Of, l.Of,
0.Of);
verts[8] = new CustomVertex.PositionTextured(-1.0f, -l.Of, -l.Of, O.Of,
l.Of);
verts[9j - new CustomVertex.PositionTextured(-1.0f, -l.Of, -l.Of, O.Of,
l.Of);
verts[10] = new CustomVertex.PositionTextured(l.Of, l.Of, -l.Of, l.Of,
0.Of);
verts[11] = new CustomVertex.PositionTextured(l.Of, -l.Of, -l.Of, l.Of,
l.Of);
```

```
// Top face
verts[12] = new CustomVertex.PositionTextured(-1.0f, 1.0f, 1.0f, 0.0f,
0.0f);
verts[13] = new CustomVertex.PositionTextured(1.0f, 1.0f, -1.0f, 1.0f,
1.0f);
verts[14] = new CustomVertex.PositionTextured(-1.0f, 1.0f, -1.0f, 0.0f,
1.0f);
verts[15] = new CustomVertex.PositionTextured(-1.0f, 1.0f, 1.0f, 0.0f,
0.0f);
verts[16] = new CustomVertex.PositionTextured(1.0f, 1.0f, 1.0f, 1.0f,
0.0f);
verts[17] = new CustomVertex.PositionTextured(1.0f, 1.0f, -1.0f, 1.0f,
1.0f);
// Bottom face (remember this is facing *away* from the camera, so vertices
   should be
   clockwise order)
verts[18] = new CustomVertex.PositionTextured(-1.0f, -1.0f, 1.0f, 0.0f,
0.0f);
verts[19] = new CustomVertex.PositionTextured(-1.0f, -1.0f, -1.0f, 0.0f,
1.0f);
verts[20] = new CustomVertex.PositionTextured(1.0f, -1.0f, -1.0f, 1.0f,
1.0f);
verts[21] = new CustomVertex.PositionTextured(-1.0f, -1.0f, 1.0f, 0.0f,
0.0f);
verts[22] = new CustomVertex.PositionTextured(1.0f, -1.0f, -1.0f, 1.0f,
1.0f);
verts[23] = new CustomVertex.PositionTextured(1.0f, -1.0f, 1.0f, 1.0f,
0.0f);
// Left face
verts[24] = new CustomVertex.PositionTextured(-1.0f, 1.0f, 1.0f, 0.0f,
0.0f);
verts[25] = new CustomVertex.PositionTextured(-1.0f, -1.0f, -1.0f, 1.0f,
1.0f);
verts[26] = new CustomVertex.PositionTextured(-1.0f, -1.0f, 1.0f, 1.0f,
0.0f);
verts[27] = new CustomVertex.PositionTextured(-1.0f, 1.0f, -1.0f, 0.0f,
1.0f);
verts[28] = new CustomVertex.PositionTextured(-1.0f, -1.0f, -1.0f, 1.0f,
1.0f);
verts[29] = new CustomVertex.PositionTextured(-1.0f, 1.0f, 1.0f, 0.0f,
0.0f);
// Right face (remember this is facing *away* from the camera, so vertices
   should be
   clockwise order)
verts[30] = new CustomVertex.PositionTextured(1.0f, 1.0f, 1.0f, 0.0f,
0.0f);
```

```
verts[31] = new CustomVertex.PositionTextured(1.0f, -1.Of, 1.Of, 1.Of,
O.Of);
verts[32] = new CustomVertex.PositionTextured(1.0f, -1.0f, -1.0f, 1.0f,
1.0f);
verts[33] = new CustomVertex.PositionTextured(1.0f, 1.0f, -1.0f, 0.0f,
1.0f);
verts[34] = new CustomVertex.PositionTextured(1.0f, 1.0f, 1.0f, 0.0f,
0.0f);
verts[35] = new CustomVertex.PositionTextured(1.0f, -1.0f, -1.0f, 1.0f,
1.0f);
buffer.SetData(verts, 0, LockFlags.None);
```

Очевидно, что самое большое изменение здесь — тип данных, используемый для сохранения списка вершин. Последние два значения с плавающей точкой, сохраненные в каждой вершине, представляют собой U и V координаты в текстуре, используемой для отображения примитива. Поскольку каждая грань куба представляет собой квадрат, текстуры будут также квадратными, таким образом, имеет смысл отображать каждый квадрат непосредственно в текстуре. Обратите внимание, что вершина в верхнем левом углу соответствует техселу 0,0, а вершина в правой нижней части отображает непосредственно тексел 1,1. При отображении каждая сторона квадрата будет включать текстуру целиком. Для того чтобы приложение выполнялось без прерываний, необходимо изменить и задать тип данных вершинного буфера:

```
vb = new VertexBuffer(typeof(CustomVertex.PositionTextured), 36, device,
    Usage.Dynamic
    j Usage.WriteOnly, CustomVertex.PositionTextured.Format, Pool.Default);
```

Давайте немного упростим процедуру рисования наших кубов, добавив следующий код:

```
private void DrawBox(float yaw, float pitch, float roll, float x, float y,
    float z,
    Texture t)
 (
    angle += 0.01f;
    device.Transform.World = Matrix.RotationYawPitchRol(yaw, pitch, roll) *
    Matrix.Translation(x, y, z);
    device.SetTexture(0, t);
    device.DrawPrimitives(PrimitiveType.TriangleList, 0, 12);
```

Первые шесть параметров этой функции те же самые, что мы использовали в начале. Мы приводим отклонение (yaw), шаг (pitch) и наклон

(roll) для нашего вращения куба, плюс X, Y, и Z для перемещения куба. Последний параметр texture t новый для нас, хотя он представляет текстуру для рендеринга нашего объекта. Мы также вызываем метод SetTexture для устройства, чтобы сообщить приложению Direct3D, какую текстуру мы хотим использовать при рендеринге этого примитива. Первый параметр этого метода определяет стадию, на которой мы управляем текстурой. Если вспомнить, мы уже упоминали о возможности отображения до восьми текстур для примитива. Таким образом, этот первый параметр является индексом текстур. Поскольку сейчас мы имеем набор координат одной текстуры, мы будем всегда использовать в качестве первого индекса значение 0. Также обратите внимание, что, поскольку мы изменяем значение «угла» поворота и осуществляем преобразование координат, вы можете удалить соответствующие строки из метода SetupCamera.

Прежде чем изменить наш код рендеринга для вызова новой функции, мы сначала должны определить те текстуры, которые будем использовать. Демонстрационная программа, имеющаяся на CD диске, включает в себя три текстуры — puck.bmp, ground.bmp, и banana.bmp, приложенные в качестве ресурсов в исходном проекте. В меню проекта нажмите «Add Existing Item» и добавьте три изображения. После этого необходимо рассмотреть свойства каждого и изменить пункт «Build Action» (Компоновка) на «Embedded Resource» (Вложенный Ресурс). Теперь мы должны объявить переменные для наших текстур, добавив их после создания вершинного буфера:

```
private Texture tex = null;
private Texture texl = null;
private Texture tex2 = null;
```

Эти текстуры мы будем использовать для рисования нашего объекта. Необходимо установить их значения до использования трех упомянутых точечных рисунков, которые мы определили как ресурсы. Добавьте следующий код сразу после создания вершинного буфера и заполнения данных:

```
tex = new Texture(device, new Bitmap(this.GetIype(), "puck.bmp"), 0,
Pool.Managed);
texl = new Texture(device, new Bitmap(this.GetType(), "banana.bmp"), 0,
Pool.Managed);
tex2 = new Texture(device, new Bitmap(this.GetType(), "ground.bmp"), 0,
Pool.Managed);
```

Данный метод для нашей текстуры включает четыре параметра. Первый — device, устройство, которое мы будем использовать для выполне-

ния текстуры. Все ресурсы (текстуры, вершинные буферы, и пр.) в сцене будут связаны с устройством. Следующий параметр — System.Drawing.Bitmap — мы будет использовать для получения данных для этой текстуры. В этом случае мы используем конструктор bitmap, чтобы загрузить наш файл из ресурсов. Третий используемый параметр мы уже кратко обсуждали при описании вершинных буферов, в нашем примере мы присвоили ему значение «0» и пока не используем. Последний параметр — пул памяти, используемый для хранения текстуры. Для удобства будем использовать управляемый пул. Ниже приведены другие конструкторы или методы, доступные для текстур:

```
public Texture ( System.IntPtr Ip , Microsoft.DirectX.Direct3D.Device
    device ,
    Microsoft.DirectX.Direct3D.Pool pool )
public Texture ( Microsoft.DirectX.Direct3D.Device device , System.Int32
    width ,
    System.Int32 height , System.Int32 numLevels ,
    Microsoft.DirectX.Direct3D.Usage usage ,
    Microsoft.DirectX.Direct3D.Format format ,
    Microsoft.DirectX.Direct3D.Pool pool )
public Texture ( Microsoft.DirectX.Direct3D.Device device ,
    System.IO.Stream data ,
    Microsoft.DirectX.Direct3D.Usage usage , Microsoft.DirectX.Direct3D.Pool
    pool )
```

Первый конструктор принимает указатель IntPtr, который является неуправляемым указателем COM интерфейса для IDIRECT3DTEXTURE9. Такой шаг используется для достижения функциональной совместимости с неуправляемым кодом. Следующий конструктор позволяет создавать пустую текстуру («blank» texture), определяя высоту, ширину и значение уровня детализации. Это иногда более предпочтительно, чем считывание этих значений из файла. Последний конструктор напоминает тот, который мы использовали в нашем приложении, только использует поток данных быстрее, чем растровый bitmap объект. Данный поток должен иметь возможность загружаться в объект System.Drawing.Bitmap для работы этого конструктора. Существуют и другие интересные функции для загрузки текстуры в класс TextureLoader, это мы рассмотрим в последующих главах.

Теперь, когда мы определили наши три текстуры и загрузили точечные рисунки, сохраненные в наших ресурсах, давайте изменим наш код вызова рисунка, чтобы отобразить текстурированные кубические объекты. Мы можем избавиться от всей процедуры выполнения рисунка для нашей функции рендеринга и заменить ее следующим:

```
// Draw our boxes
DrawBox(angle / (float)Math.PI, angle / (float)Math.PI * 2.0f, angle /
    (float)Math.PI /
    4.0f, 0.0f, 0.0f, 0.0f, tex);
'DrawBox(angle / (float)Math.PI, angle / (float)Math.PI / 2.0f, angle /
    (float)Math.PI *
    4.0f, 5.0f, 0.0f, 0.0f, tex1);
DrawBox(angle / (float)Math.PI, angle / (float)Math.PI * 4.0f, angle /
    (float)Math.PI /
    2.0f, -5.0f, 0.0f, 0.0f, tex2);
```

Данная процедура должна отобразить каждый из трех наших имеющихся кубических объектов, где вместо граней, окрашенных в свои цвета, мы должны получить текстурированные объекты. Типовая программа, включенная в CD диск, отобразит девять текстурированных кубов, **рис.3.3.**

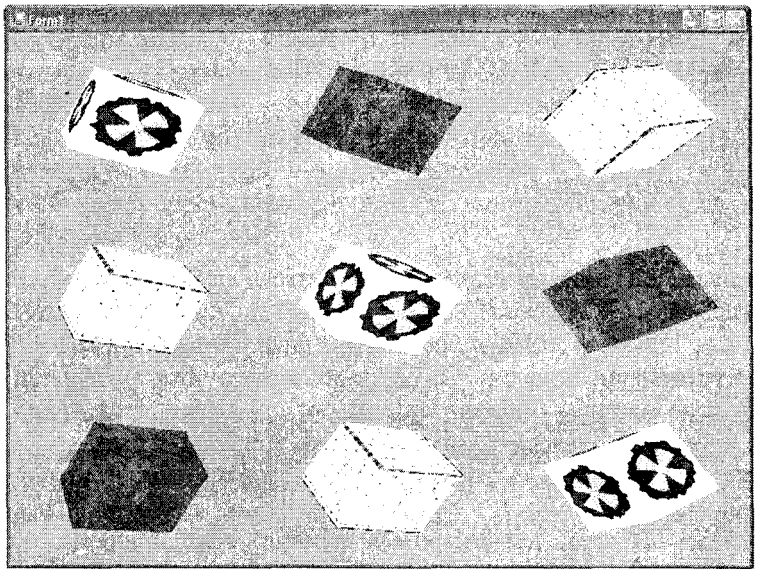


Рис. 3.3. Текструрированные кубы

Краткие выводы

В этой главе мы применили более эффективные методы отображения к нашим приложениям и использовали процедуру рендеринга, а также.

Использовали вершинные буферы для сохранения данных вершин, и нарисовали несколько примитивов из того же самого буфера.

- Создали текстуры для более реалистичного отображения объектов.
- Включили данные в наш вершинный буфер и отобразили текстуры на примитивах.

В следующей главе мы обсудим более совершенные методы технологии рендеринга, включая использование индексных буферов, других типов примитивов и использование буферов глубины (или Z-буферов).

Глава 4. Более совершенные технологии рендеринга

Теперь, когда мы ознакомились с основными принципами технологии рендеринга, мы можем усовершенствовать методы отображения рисунков. В этой главе мы рассмотрим следующее.

- Использование различных примитивов, отличающихся от набора треугольников.
- Использование индексных буферов для более управляемого рендеринга.
  Использование буферов глубины — Z-буферов для более реалистичного рендеринга.

Рендеринг с использованием примитивов различного типа

До сих пор мы рассматривали в качестве типа примитива только набор треугольников. Имеются и другие типы, которые мы можем использовать для рисования.

- PointList — набор точек, тип примитива с самоописанием, отображает на экране объект из набора точек. Вы не можете использовать этот тип, когда рисуете индексированные примитивы (которые мы рассмотрим позже в этой главе), **рис.4.1**
  LineList — набор линий, отображает каждую пару вершин как отдельную линию. При использовании этого типа примитива необходимо иметь для соединения четное число вершин (по крайней мере, две), **рис.4.2.**
- LineStrip — набор ломаных линий, отображает вершины в виде отдельной ломаной линии. После того как прорисовывается первая линия, замыкающая первую пару вершин, каждая последующая линия рисуется из последней точки предыдущей линии. Вам нужно иметь по крайней мере две вершины при использовании этого типа примитива, **рис.**4.3
- TriangleList—набор треугольников, данный примитив мы использовали до настоящего момента. Отображает набор трех вершин как отдельный, изолированный треугольник. Отбор невидимой поверхности определяется текущим состоянием рендера отбора, **рис.4.4.**

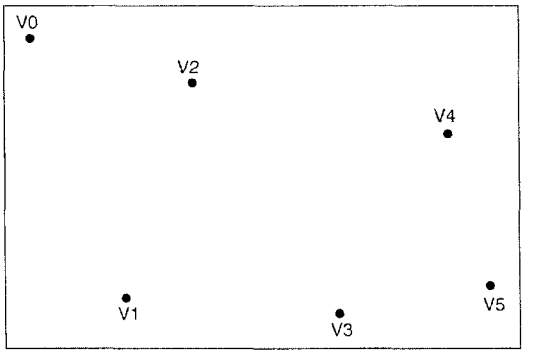


Рис. 4.1. Набор точек

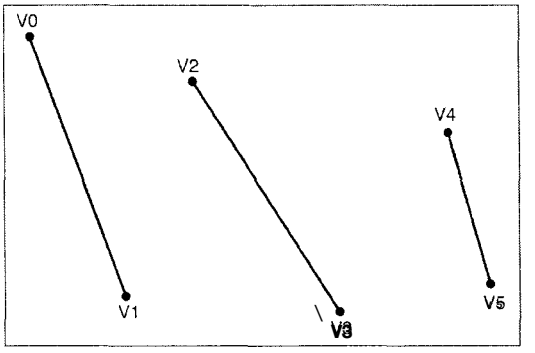


Рис. 4.2. Набор линий

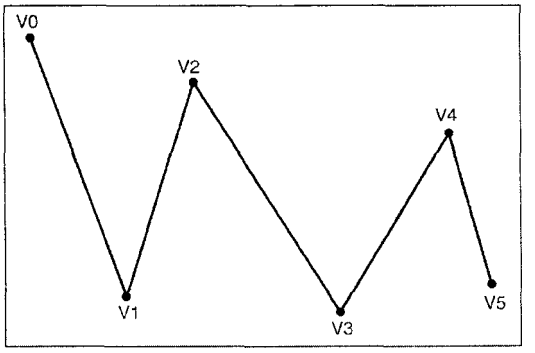


Рис. 4.3. Набор ломаных линий

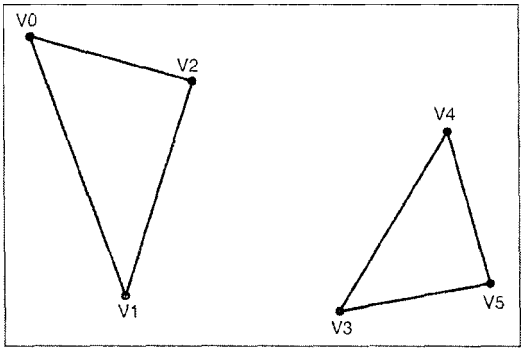


Рис. 4.4. Набор треугольников

TriangleStrip — полоска из треугольников, рисует каждый следующий треугольник, используя при построении две имеющиеся вершины предыдущего треугольника. Режим отбора автоматически отражает изображение всех четных треугольниках. Поскольку данный тип использует последние две вершины предыдущего треугольника, вершина полученного треугольника будет расположена с противоположной стороны от соединяющей эти вершины линии. Это наиболее распространенный тип примитива для сложных трехмерных объектов, **рис.4.5.**

TriangleFan — веер из треугольников, похож на предыдущий тип TriangleStrip, за исключением того, что все треугольники используют одну общую вершину, **рис.4.6.**

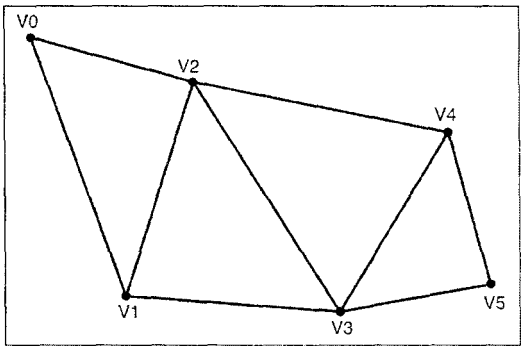


Рис. 4.5. Полоска из треугольников

При рисовании мы можем использовать одни и те же данные о вершинах для любого из этих типов примитивов. Приложение Direct3D интерпретирует данные вершин каждый раз по-разному, в зависимости от типа примитива, который вы ему сообщаете. Давайте запишем «быстрый бит» кода для рисования различных типов примитивов.

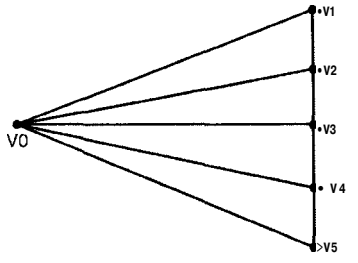


Рис. 4.6. Веер из треугольников

Типовая программа, включенная в CD диск, использует вершинный буфер, который мы создавали в главе 3. Отметим изменения в этой программе. Поскольку этот код не предусматривает перемещения вершин, функция преобразования в методе SetupCamera была удалена, также как и все ссылки на значение угла поворота «angle». Затем была добавлена следующая константа, определяющая число элементов:

```
private const int NumberItems = 12;
```

Значение «12» выбрано произвольно, но с учетом некоторых условий. Слишком много вершин выглядело бы сейчас громоздко на экране, однако, все-таки желательно иметь достаточное число вершин, а также чтобы оно было четным и делилось на три. В этом случае мы можем корректно использовать примитивы LineList и TriangleList. Затем необходимо изменить алгоритм создания вершинного буфера и функцию OnVertexBufferCreate, как приведено в листинге 4.1:

Листинг 4.1. Создание случайных вершин.

```
vb = new VertexBuffer(typeof(CustomVertex.PositionColored), Numberltems,
   device,
   Usage.Dynamic \ Usage.WriteOnly, CustomVertex.PositionColored.Format,
   Pool.Default);
// Create a list of vertices equal to the number of items
CustomVertex.PositionColored[] verts = new
   CustomVertex.PositionColored[Numberltems];
7 Randomly select the position of this vertex.. Keep it within a range of
   5.
   for(int i = 0; i < Numberltems; i++)
```

```
    float xPos = (float)(Rnd.NextDouble() * 5.Of) - (float)(Rnd.NextDouble()
    * 5.0f);
    float y?os = (float) (Rnd.NextDouble() * 5.Of) - (float) (Rnd.NextDouble()
    * 5.Of);
   .float zPos = (float) (Rnd.NextDouble() * 5.Of) - (float) (Rnd.NextDouble()
    * 5.Of);
    verts[i].SetPosition(new Vector3(xPos, yPos, zPos));
    verts[i] .Color = RandomColor.ToArgb();
1
buffer.SetData(verts, 0, LockFlags.None);
```

Здесь нет ничего необычного. Мы изменили наш вершинный буфер, чтобы установить корректное число вершин (определяемое константой Numberltems). Затем мы изменили нашу функцию создания вершин, чтобы расположить значения вершин случайным образом. Вы можете найти описание параметров «Rnd» и «RandomColor» в исходном тексте на CD диске

Затем необходимо изменить вызов рисунка. Простой способ показать, как эти различные типы примитивов взаимодействуют между собой, заключается в том, чтобы составить последовательность команд, позволяющую просматривать их друг за другом. Это можно организовать, задав временную задержку между текущим временем (в тактах) и временем запуска. Добавьте следующие значения переменных в раздел объявления типов данных:

```
private bool needRecreate = false;
// Timing information
private static readonly int InitiallickCount =
System.Environment.TickCount;
```

Данная булева переменная позволит нам перезаписывать наш вершинный буфер в начале каждого цикла, что позволить нам не отображать многократно одну и ту же вершину. Замените обращение к функции DrawPrimitives кодом, приведенным в листинге 4.2:

Листинг 4.2. Рендеринг различных типов примитивов.

```
// We will decide what to draw based on primitives
int index = ((System.Environment.TickCount - InitialTickCount) / 2000) \ 6;
switch (index)
 {
case 0: // Points
   device.DrawPrimitives(Primitivelype.PointList, 0, Numberltems);
   if (needRecreate)
    }
```

```
        // After the primitives have been drawn, recreate a new set
        OnVertexBufferCreate(vb,  null);
        needRecreate = false;
    }
    break;
case 1: // LineList
    device.DrawPrimitives(PrimitiveType.LineList, 0, NumberItems / 2);
    needRecreate = true;
    break;
case 2: // LineStrip
    device.DrawPrimitives(PrimitiveType.LineStrip, 0, NumberItems - 1);
    break;
case 3: // TriangleList
    device.DrawPrimitives(PrimitiveType.TriangleList, 0, NumberItems / 3);
    break;
case 4: // TriangleStrip
    device.DrawPrimitives(PrimitiveType.TriangleStrip, 0, NumberItems - 2);
    break;
case 5: // TriangleFan
    device.DrawPrimitives(PrimitiveType.TriangleFan, 0, NumberItems - 2);
    break;
}
```

Это относительно очевидный алгоритм. В зависимости от того, какой тип примитива мы выбираем в нашем цикле, мы просто вызываем функцию DrawPrimitives с соответствующим типом примитива. Обратите внимание, что для набора точек мы просто используем число вершин в качестве числа используемых точек. Описанные выше LineList и TriangleList отображают изолированные примитивы, которые для нашего обращения к DrawPrimitives с этими типами должны иметь число вершин, кратных числу вершин в отдельном примитиве (два или три, соответственно). Учитывая тот факт, что каждая последующая линия рисуется из последней точки предыдущей линии, количество отображаемых линий для примитивов LineStrip должно быть на одну меньше, чем текущее число вершин. Для примитивов TriangleStrip и TriangleFan это требование похоже, с поправкой на то, что число треугольников должно быть на два меньше, чем общее число вершин.

Выполнение данного приложения отобразит сначала последовательность точек на экране, а затем соответствующие линии. Затем эти строки соединятся в сплошную ломаную линию, а далее в набор изолированных треугольников. В конечном счете, на экране отобразится полоса или, в другом случае, веер из треугольников. Обратите внимание, размеры точек можно изменять (значение «scale» в установках состояния рендера). Добавьте следующую строку к вашей функции SetupCamera для того, чтобы увеличить размер каждой точки в три раза:

```
device.RenderState.PointSize = 3.0 f ;
```

Можно заметить, что точки значительно больше, чем они были первоначально.

Теперь, когда у нас имеются различные типы примитивов, можно попробовать оптимизировать объем используемой памяти для наших данных.

Использование индексных буферов

Если вспомнить наше первое приложение, где мы рисовали куб и создавали данные для 36 вершин, мы имели 2 треугольника для каждой из шести граней куба; таким образом, мы имели 12 примитивов. Поскольку каждый примитив имеет 3 вершины, общее число вершин примитивов равно 36. Однако, в действительности вершин было только 8, для каждого из углов куба.

Сохранение данных одних и тех же вершин для приведенного примера может показаться не таким убедительным, но в больших приложениях, где вы имеете огромное количество данных, было бы весьма полезным не сохранять многократно одинаковые вершины, экономя при этом место в памяти. К счастью, Direct3D имеет механизм для решения таких задач, называемый «индексным буфером».

Исходя из названия, индексный буфер — это буфер для записи индексов в данные вершин. Индексы, сохраненные в этом буфере, могут быть 32-разрядными (целочисленные данные) или 16-разрядными (короткие данные). При небольшом количестве индексов можно использовать 16-разрядный формат.

При использовании индексного буфера для отображения примитивов каждый индекс в буфере соответствует конкретной вершине и ее данным. Например, треугольник с индексами 0, 1, 6 отобразился бы в соответствии с данными, сохраненными в этих вершинах. Давайте изменим наше приложение, рисующее куб, используя при этом индексы. Сначала изменим функцию создания данных вершин, как показано в листинге 4.3:

Листинг 4.3. Создание вершин для нашего куба.

```
vb = new VertexBuffer(typeof(CustomVertex.PositionColored), 8, device,
   Usage.Dynamic |
   Usage.WriteOnly, CustomVertex.PositionColored.Format, Pool.Default);
CustomVertex.PositionColored[] verts = new CustomVertex.PositionColored[8];
// Vertices
verts[0] = new CustomVertex.PositionColored(-1.0f, 1.0f, 1.0f,
Color.Purple.ToArgb());
```

```
verts[1] = new CustomVertex.PositionColored(-1.0f, -1.0f, 1.0f,
Color. Red. ToArgb());
verts[2] = new CustomVertex.PositionColored(1.0f, 1.0f, 1.0f,
Color.Blue.ToArgb());
verts[3] = new CustomVertex.PositionColored(1.0f, -1.0f, 1.0f,
Color.Yellow.ToArgb());
verts[4] = new CustomVertex.PositionColored(-1.0f, 1.0f, -1.0f,
Color.Gold.ToArgb());
verts[5] = new CustomVertex.PositionColored(1.0f, 1.0f, -1.0f,
Color.Green.ToArgb());
verts[6] = new CustomVertex.PositionColored(-1.0f, -1.0f, -1.0f,
Color.Black.ToArgb());
verts[7] = new CustomVertex.PositionColored(1.0f,-1.0f,-1.0f,
Color.WhiteSmoke.ToArgb());
buffer.SetData(verts, 0, LockFlags.None);
```

Как можно увидеть, мы значительно уменьшили количество вершин, оставив только те 8, которые составляют углы куба. Мы можем нарисовать 36 вершин, задавая различные параметры для каждого набора из 8 вершин. Но, исходя из предыдущего приложения, мы можем найти каждую из используемых 36 вершин и соответствующий ей индекс в нашем новом списке. Добавьте список индексов, приведенный в листинге 4.4, к разделу определения данных:

Листинг 4.4. Данные индексного буфера для создания куба.

```
private static readonly short[] indices = {
   0,1,2, // Front Face
   1,3,2, // Front Face
   4,5,6, // Back Face
   6,5,7, // Back Face
   0,5,4, // Top Face
   0,2,5, // Top Face
   1,6,7, // Bottom Face
   1,7,3, // Bottom Face
   0,6,1, // Left Face
   4,6,0, // Left Face
   2,3,7, // Right Face
   5,2,7 // Right Face
   };
```

Для простоты чтения индексный список разбивается на три — для каждого треугольника, которогое мы будем рисовать. Ясно, что лицевая сторона куба создается из двух треугольников. Первый треугольник использует вершину О, 1, 2, в то время как второй треугольник использует

вершину 1, 3, 2. Точно так же для правой грани куба первый треугольник использует вершину *2,* 3, 7, второй треугольник использует 5, 2, 7. Правила отбора невидимой поверхности остаются в силе и при использовании индексов.

Тем не менее, имея только список индексов, мы вряд ли сможем что-то сделать, не изменив наше приложение. Для этого необходимо создать индексный буфер. Добавьте следующую строку после объявления вершинного буфера:

```
private IndexBuffer ib = null;
```

Этот объект будет использоваться и для хранения индексов, и для обеспечения доступа приложения Direct3D к этим индексам. Данная процедура напоминает использование вершинного буфера, который мы уже создали, только вместо данных о вершине IndexBuffer содержит индексы. Рассмотрим примеры использования данного объекта, заполнив его данными. После создания вершинного буфера добавьте код, приведенный в листинге 4.5:

Листинг 4.5. Создание индексного буфера.

```
ib = new IndexBuffer(typeof(short),
indices.Length,device,Usage.WriteOnly,Pool.Default);
ib.Created += new EventHandler(this.OnIndexBufferCreate);
OnIndexBufferCreate(ib, null);
private void OnIndexBufferCreate(object sender, EventArgs e)

    IndexBuffer buffer = (IndexBuffer)sender;
    buffer.SetData(indices, 0, LockFlags.None);
}
```

Обратите внимание, что конструктор для индексного буфера напоминает конструктор для вершинного буфера. Единственное различие — ограничения на параметры типа. Как упоминалось выше, можно использовать как короткие данные (System.Int16), так и целочисленные значения (System.Int32) данных. При использовании индексного буфера мы также вызываем обработчик события и функцию обработчика прерываний для первого запуска. Затем мы просто заполняем наш индексный буфер необходимыми данными.

Теперь, чтобы использовать введенные данные, необходимо только вставить код рендеринга. Если вы помните, была функция, именуемая SetStreamSource, которая сообщала приложению Direct3D, какой вершинный буфер будет использоваться при выполнении рендеринга. Существует похожая функция и для индексных буферов, однако в этот раз, она ис-

пользуется просто как признак, поскольку одновременно использоваться может только один тип индексного буфера. Установите этот признак сразу после вызова функции SetStreamSource:

```
device.Indices = ib;
```

Теперь, когда Direct3D «знает» о нашем индексном буфере, мы должны изменить вызов рисунка. В данном случае мы пытаемся отобразить 12 примитивов (36 вершин) из нашего вершинного буфера, который естественно будет работать некорректно, поскольку он включает данные только о восьми вершинах. Следует вернуться назад и добавить функцию DrawBox:

```
private void DrawBox(float yaw, float pitch, float roil, float x, float y,
float z)
(
   angle += O.Olf;
   device.Transform.World = Matrix.RotationYawPitchRolI(yaw, pitch, roll) *
      Matrix.Translation(x, y, z);
   device.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0, 8, 0,
      indices.Length / 3);
1
```

Итак, мы изменили вызов процедуры создания рисунка DrawPrimitives на DrawIndexedPrimitives. Рассмотрим прототип этой функции:

```
public void DrawIndexedPrimitives (
   Microsoft.DirectX.Direct3D.PrimitiveType primitiveType
   , System.Int32 baseVertex , System.Int32 minVertexIndex , System.Int32
   numVertices ,
   System.Int32 startlndex , System.Int32 primCount )
```

Первый параметр такой же, как и в предыдущей функции, — тип примитивов, которые мы собираемся рисовать. Параметр BaseVertex — смещение от начала индексного буфера до первого индекса вершины. Параметр MinVertexIndex — минимальный индекс вершины в этом вызове. Параметр NumVertices — значение очевидно (число вершин, используемых в течение этого вызова), однако он запускается вместе с параметрами baseVertex и minVertexIndex. Параметр StartIndex определяет местоположение в массиве для запуска считывания данных о вершине. Последний параметр остается тем же самым (число отображаемых примитивов).

Таким образом, видно, что мы пытаемся нарисовать 8 вершин, используя индексный буфер, для того чтобы отобразить 12 примитивов для нашего куба. Теперь давайте удалим функции DrawPrimitives и заменим их

следующими строками для нашего метода DrawBox, приведенного в листинге 4.6:

Листинг 4.6. Рисование кубов.

```
// Draw our boxes
DrawBoxfangle / (float)Math.PI, angle / (float)Math.PI * 2.0f, angle /
    (float)Math.PI /
   4.0f, 0.0f, 0.0f, 0.0f);
DrawBoxfangle / (float(Math.PI, angle / (float)Math.PI / 2.0f, angle /
    (float)Math.PI *
   4.0f, 5.0f, 0.0f, 0.0f);
DrawBoxfangle / (float)Math.PI, angle / (float)Math.PI * 4.0f, angle /
    (float)Math.PI /
   2.0f, -5.0f, 0.0f, 0.0f);
DrawBoxfangle / (float(Math.PI, angle / (float)Math.PI * 2.0f, angle /
    (float)Math.PI /
   4.0f, 0.0f, -5.0f, 0.0f);
DrawBoxfangle / (float)Math.PI, angle / (float)Math.PI / 2.0f, angle /
    (float)Math.PI *
   4.0f, 5.0f, -5.0f, 0.0f);
DrawBoxfangle / (float)Math.PI, angle / (float)Math.PI * 4.0f, angle /
    (float)Math.PI /
   2.0f, -5.0f, -5.0f, 0.0f);
DrawBoxfangle / (float)Math.PI, angle / (float)Math.PI * 2.0f, angle /
    (float)Math.PI /
   4.0f, 0.0f, 5.0f, 0.0f);

DrawBoxfangle / (float)Math.PI, angle / (float)Math.PI / 2.0f, angle /
    (float)Math.PI *
   4.0f, 5.0f, 5.0f, 0.0f);
DrawBoxfangle / (float)Math.PI, angle / (float)Math.PI * 4.0f, angle /
    (float)Math.PI /
   2.0f, -5.0f, 5.0f, 0.0f);
```

При выполнении данное приложение отобразит на экране очень красочные вращающиеся кубы.

Причиной, по которой каждая вершина в нашем списке имеет разные цвета, является необходимость визуально показать один из недостатков использования индексных буферов при одновременном использовании вершин несколькими примитивами. Когда вершина разделена «между» несколькими примитивами, все данные вершин разделены, включая цвета и данные нормали. Таким образом, вы можете либо действительно одновременно использовать эту вершину, либо вам необходимо сначала определить, может ли одновременное использование этих данных вызы-

вать ошибки при освещении объекта (поскольку при вычислении освещения используются данные нормалей к поверхностям). Вы можете так же видеть, что каждый угол куба представлен цветом его вершины, а грани куба интерполированы из цветов его вершин.

Использование буферов глубины или Z-буферов

Буфер глубины, ***depth buffer*** (часто упоминаемый как Z-буфер или W-буфер), используется приложением Direct3D для хранения информации о глубине отображаемого объекта. Эта информация используется в процессе растеризации, чтобы определить, насколько пиксели перекрывают друг друга. На данном этапе, наше приложение не имеет буфера глубины, поэтому пиксели изображения не перекрываются в течение всего процесса растеризации. Попробуем нарисовать еще несколько кубов, которые будут накладываться на некоторые из имеющихся. Добавьте следующие строки в конце наших существующих обращений к DrawBox:

```
DrawBox(angle / (float)Math.PI, angle / (float)Math.PI * 2.0f, angle /
    (float)Math.PI /
    4.0f, 0.0f, (float)Math.Cos(angle), (float)Math.Sin(angle));
DrawBox(angle / (float)Math.PI, angle / (float)Math.PI / 2.0f, angle /
    (float)Math.PI *
    4.0f, 5.0f, (float)Math.Sin(angle), (float)Math.Cos(angle));
DrawBox(angle / (float)Math.PI, angle / (float)Math.PI / 2.0f, angle /
    (float)Math.PI /
    2.0f, -5.0f, (float)Math.Cos(angle), (float)Math.Sin(angle) );
```

Все, что мы пытаемся здесь сделать — это вращать три куба вокруг трех центральных кубов, расположенных по горизонтали. Выполняя это приложение, можно увидеть, как кубы перекрываются, но четко отследить границы между кубами невозможно. Кубы кажутся сплошными объектами. Вот здесь как раз и пригодится буфер глубины.

Добавление буфера глубины к нашему приложению является достаточно простой задачей. Можно вспомнить параметры представления или вида, которые мы пересылали в наш конструктор устройства для включения информации о буфере глубины. Появились два новых параметра, которые мы должны заполнить, чтобы включить буфер глубины в наше устройство:

```
public Microsoft.DirectX.Direct3D.DepthFormat AutoDepthStencilFormat [get,
set]
public bool EnableAutoDepthStencil [get, set]
```

ОПТИМИЗАЦИЯ ПАМЯТИ ПРИ ИСПОЛЬЗОВАНИИ ИНДЕКСНЫХ БУФЕРОВ

Для того чтобы определить, насколько использование индексного буфера позволяет экономить память для нашего приложения, сравним относительно простые приложения отображения куба с использованием индексного буфера и без него. В первом сценарии мы создавали вершинный буфер, включающий в себя 32 вершины типа CustomVertex.PositionColored. Эта структура содержала 16 байт (4 байта на каждый параметр X, Y, Z и цвет). Можно умножить этот размер на число вершин, получается, что наши данные вершин занимают 576 байт.

Теперь сравним это с алгоритмом, использующим индексный буфер. Мы используем только 8 вершин (того же самого типа), так что наш размер данных вершин составляет 128 байтов. Однако, мы также должны хранить наши индексные данные, при этом мы используем короткие индексы (2 байта на каждый), 36 индексов. Таким образом, наши индексные данные занимают 72 байта, суммируя это значение с 128 байтами, мы имеем полный размер 200 байтов.

Сравните это с нашим первоначальным размером 576 байтов. Мы сократили требуемую память на 65 %. Экстраполируя эти значения на очень большие сцены, нетрудно представить, насколько эффективным может быть использование индексных буферов в плане экономии памяти.

Установка EnableAutoDepthStencil в значение «true» включает буфер глубины для устройства, используя необходимый формат глубины, указанный в параметре AutoDepthStencilFormat. Применимые значения форматов глубины DepthFormat перечислены в таблице 4.1:

Таблица 4.1. Возможные форматы Z буферов

Формат	**Описание**
D16	16-разрядный Z-буфер
D32	32-разрядный Z-буфер
D16Lockable	16-разрядный Z-буфер с возможностью блокировки
D32FLockable	32-разрядный Z-буфер. Блокируемый формат. Использует стандарт ШЕЕ с плавающей запятой
D15S1	16-разрядный Z-буфер, использует 15 бит на канал глубины, с последним битом, используемым для шаблонного (stencil) канала (такие каналы будут обсуждаться позднее)

Формат	Описание
D24S8	32-разрядный Z-буфер. Использует 24 бита для канала глубины и 8 бит для шаблонного канала
D24X8	32-разрядный Z-буфер. Использует 24 бита для канала глубины, оставшиеся 8 бит игнорируются
D24X4S4	32-разрядный Z-буфер. Использует 24 бита для канала глубины, с 4 битами, используемыми для шаблонного канала. Оставшиеся 4 бита игнорируются
D24FS8	32-разрядный Z-буфер. Использует 24 бита для канала глубины (с плавающей запятой) и 8 битов для шаблонного канала

ВЫБОР СООТВЕТСТВУЮЩЕГО БУФЕРА ГЛУБИНЫ

На сегодняшний день практически любая приобретаемая графическая плата поддерживает Z-буфер; однако, в зависимости от обстоятельств, при использовании этого буфера могут возникнуть проблемы. При вычислении глубины пиксела приложение Direct3D размещает пиксель в определенном диапазоне Z-буфера (обычно от O.Of до 1 .Of), но это размещение редко является равномерным по всему диапазону. Отношение передней и задней плоскости напрямую определяет картину распределения в Z-буфере.

Например, если ваша передняя плоскость определяется значением 1 .Of, а задняя плоскость значением 100.Of, то 90 % всего диапазона будут использоваться в первых 10 % вашего буфера глубины. В предыдущем примере, если бы задняя плоскость определялась значением 1000.Of, то 98 % всего диапазона использовалось бы в первых 2 % буфера глубины. Это могло бы привести к появлению «артефактов» при отображении отдаленных объектов.

Использование другого буфера глубины — W-буфера — устраняет эту проблему, но имеет свои собственные недостатки. При использовании W-буфера возможно появление «артефактов» скорее для ближних объектов, нежели для отдаленных. Также следует отметить, что не так много графических карт поддерживают W-буферы, по сравнению с Z-буферами.

И последнее, для увеличения эффективности процедуры рендеринга при использовании буферов глубины лучше отображать элементы от передней плоскости (самое большое значение Z) к задней (самое маленькое значение Z). В течение растеризации сцены Direct3D может быстро отбросить пиксель, который уже перекрыт, и отобразить видимое поле рисунка полностью. Это не относиться к случаю, когда мы проводим рендеринг данных с алфавитными компонентами, но к этому вопросу мы вернемся позже.

Буферы глубины большего размера могут хранить гораздо большее количество данных глубины, но ценой производительности и скорости. Если в этом нет необходимости, используйте буферы с меньшим размером. Современные графические платы поддерживают как правило 16-разрядные буферы глубины, поэтому мы можем добавить это значение к существующей структуре параметров:

```
presentParams.EnableAutoDepthStencil = true;
presentParams.AutoDepthStencilFormat = DepthFormat.D16;
```

Замечательно, теперь в нашем устройстве имеется буфер глубины. Остается опробовать работу нашего приложения с новым буфером и сравнить результаты.

Запускаем приложение.

К сожалению, что-то нарушилось в выполнении приложения, и мы не видим того, чего ожидали. Что же случилось с нашими кубами? Почему добавление буфера глубины к нашему устройству вызвало сбой в работе программы? Это достаточно интересно: оказывается, наш буфер глубины ни разу не был «очищен» или сброшен, что и вызвало ошибку в работе. Попытаемся сделать это. Измените обращение к функции clear следующим образом:

```
device.Clear(ClearFlags.Target | ClearFlags.ZBuffer, Color.CornflowerBlue,
1.0f, 0);
```

Теперь приложение работает нормально. Вы можете убедиться, что кубы разделены, и буфер глубины функционирует должным образом.

Краткие выводы

В этой главе мы охватили более совершенные методы рендеринга, включая.

- Использование индексных буферов, позволяющих обойти лишнее дублирование вершин и таким образом увеличить эффективность обработки и отображения данных.
- Различные типы примитивов, принципы создания каждого из них.
- Буферы глубины, их описание и использование.

В следующей главе мы будем отображать на экране реальные объекты, загруженные из файлов. Кроме того, пришло время переходить к объектам «mesh».

Глава 5. Рендеринг Mesh-объектов

В течение этой главы мы рассмотрим следующие пункты.

Применение объектов Mesh.

- Использование материалов, загрузка материалов.
- Рисование общих объектов с использованием Mesh-технологии.
- Использование Mesh для загрузки и рендеринга внешних файлов.

Определение Mesh-объектов

В процессе построения и отображения объектов, помимо создания вершин вручную, ввода индексных данных, возможна также загрузка данных вершин из внешнего источника, например файла. Обычный формат таких файлов — .X файл. В предыдущих главах мы создавали только простые объекты, которые отображались на экране. Определение данных для треугольника и куба не составило огромного труда для нас. Но если представить объект, у которого имеются десятки тысяч вершин, вместо 36, которых мы использовали, усилия для написания такого кода или процедуры были бы более чем значительными.

К счастью, в Управляемом DirectX есть объект, который может инкапсулировать сохраненные и загружаемые вершины, а также индексировать данные. Будем называть такой объект — объект Mesh. Объекты Mesh могут использоваться для сохранения любого типа графических данных, но главным образом предназначены для формирования сложных моделей. Технология Mesh включает несколько методов или алгоритмов, позволяющих увеличить эффективность рендеринга отображаемых объектов. Все Mesh-объекты будут содержать вершинный буфер и индексный буфер (с которыми мы уже ознакомились), плюс буфер атрибутов, который мы рассмотрим позже в этой главе.

Фактический объект Mesh постоянно находится в библиотеке расширений (Direct3D Extensions library (D3DX)). До сих пор мы имели только ссылки на главное приложение Direct3D, поэтому, прежде чем использовать Mesh-объекты для нашего проекта, мы должны добавить ссылку на библиотеку Microsoft.DirectX.Direct3DX.dll. Далее, используя объект Mesh, попробуем создать наше приложение для вращающегося куба.

После того как мы загрузили необходимые ссылки в наш проект, необходимо объявить переменную для объекта Mesh. Ее можно разместить там же, где размещались бы значения вершинного или индексного буфера:

```
private Mesh mesh = null;
```

Имеются три метода для создания объектов Mesh; однако, сейчас нам не понадобится ни один из них. Есть несколько статических методов клас-

са Mesh, которые мы можем использовать при создании или загрузке различных моделей. Один из первых методов, на который следует обратить внимание, — метод «Box». Судя по названию, данный метод создает объект Mesh, включающий в себя «куб». Для рассмотрения и применения данного метода добавим следующую строку сразу после кода создания устройства:

```
mesh = Mesh.Box(device, 2.0f, 2.0f, 2.0f);
```

Данный метод создаст новый Mesh-объект, который содержит вершину и индексы, необходимые для рендеринга куба с заданной высотой, шириной и глубиной (значение 2.0f). Это куб того же размера, что и созданный нами раньше вручную, используя вершинный буфер. Невероятно, но мы уменьшили количество строк этой процедуры до одной.

Теперь, когда мы создали Mesh-объект, будут ли наши действия теми же, что и раньше, или нам необходимы новые методы? Ранее, при отображении нашего куба мы вызывали функцию SetStreamSource, чтобы сообщить приложению Direct3D, какой вершинный буфер используется для считывания данных, а также определяли индексы и свойства формата вершин. При использовании Mesh-объектов отпадает необходимость в использовании указанных действий.

РИСУНОК, ИСПОЛЬЗУЮЩИЙ MESH-ОБЪЕКТ

Уточним некоторые вопросы относительно вершинного буфера для Mesh-объектов. При использовании объектов Mesh сохраняются вершинный буфер, индексный буфер и формат вершин. Когда Mesh-объект подвергается рендерингу, он автоматически устанавливает потоковый источник, также как индексы и свойства формата вершин.

Теперь, когда наш Mesh-объект создан, необходимо отобразить его на экране. Все Mesh-объекты разбиты на группу подмножеств (на основе буфера атрибутов, который мы обсудим вскоре), а также имеется метод DrawSubset, который мы можем использовать для нашего рендеринга. Перепишем функцию DrawBox следующим образом:

```
private void DrawBox(float yaw, float pitch, float roll, float x, float y,
float z)
{
   angle += 0.01f;
   device.Transform.World = Matrix.RotationYawPitchRoil(yaw, pitch, roll) *
      Matrix.Translation(x, y, z);
   mesh.DrawSubset(0);
}
```

Как можно видеть, мы заменили наш вызов DrawIndexedPrimitives на вызов DrawSubset. Стандартные примитивы, создаваемые классом Mesh (такие как Mesh.Box), будут всегда иметь единственное подмножество.

Это пока все, что мы должны были сделать для работы нашего приложения. Удивительно просто, не правда ли? Теперь опробуем его.

Итак, мы опять получили наши девять вращающихся кубов, но, увы, они все бесцветные. Если посмотреть на вершинный формат, созданный для объекта Mesh (через свойства VertexFormat), можно увидеть, что здесь списаны только данные нормалей и местоположение объекта. Цвета объекта Mesh не были определены и, поскольку мы выключили подсветку, кубы остались неосвещенными и неокрашенными.

Если вы помните, в главе 1 («Введение в Direct3D») говорилось о том, *что* освещение работает только тогда, когда имеются данные о нормалях, сохраненные для вершин, и поскольку у нас имеются некоторые данные о нормалях для нашего куба, попробуем включить только подсветку фона. Освещение в устройстве установлено по умолчанию, поэтому вы можете либо удалить строку, определяя значение «false», либо установить соответствующее значение «true».

Итак, мы успешно превратили наши неокрашенные кубы в черные. Но к сожалению, мы не имеем никакого освещения самой сцены. Пора ознакомиться с термином «общее освещение» или «общий свет» и использовать данную характеристику в нашей сцене.

Общий свет обеспечивается постоянным источником света для сцены. При этом все объекты в сцене будут освещены равномерно, поскольку общее освещение не зависит ни от каких коэффициентов (например, местоположения источника, направленности, ослабления и пр.), в отличие от других методов освещения. Вам даже не понадобятся данные о нормалях для задания общего света. Но, несмотря на свою эффективность, такой свет не позволяет добиться реалистичности картинки. Тем не менее, пока мы воспользуемся преимуществами общего освещения. Добавьте следующую строку в раздел, где описывалось состояние рендера подсветки:

```
device.RenderState.Ambient  =  Color.Red;
```

Общий свет и расцветка в сцене определяются состоянием рендера общего освещения. В данном случае для большей наглядности эффектов мы хотим, чтобы наш общий свет стал красным. Запуская приложение теперь, мы ждем девять вращающихся красных кубов, но, увы, они все еще черные. Что же мы еще не задали?

Использование материалов и освещения

Итак, чем же отличается нынешнее освещение от того, которое мы уже использовали раньше? Единственное главное различие (кроме того, что используется Mesh) — недостаток цвета в наших данных вершины. Это и привело к неудаче в нашем случае.

Для того чтобы Direct3D правильно вычислял цвет в отдельных точках на трехмерном объекте, не достаточно указать только цвет источника освещения, требуется указать и то, как объект будет отражать свет определенного цвета. В реальном мире, если вы освещаете красным светом синюю поверхность, на последней появится мягкий фиолетовый оттенок. Вам необходимо описать, как поверхность (в данном случае, наш куб) отражает свет от источника.

В приложении Direct3D эту особенность описывают так называемые «материалы». Вы можете определить, каким образом объект будет отражать общий диффузный свет, какие участки будут освещены больше или меньше (обсудим позже), и отражает ли объект свет вообще. Добавьте следующий код к вашему обращению DrawBox (до вызова DrawSubset):

```
Material boxMaterial = new Material!);
boxMaterial.Ambient = Color.White;
boxMaterial.Diffuse = Color.White;
device.Material = boxMaterial;
```

Здесь мы создаем новый материал и устанавливаем значения общего и диффузного освещения, выбрав белый цвет. Использование белого означает, что мы отразим весь общий и диффузный свет, падающий на объекты. Затем мы используем свойства материалов в нашем устройстве для того, чтобы Direct3D «знал», какой материал использовать при рендеринге.

Запуская приложение, мы вновь видим на экране вращающиеся красные кубы. Изменение цвета общего освещения изменит и цвет каждого куба в сцене, а изменение цвета компонента материала изменит цвет отраженного от объекта света. Изменяя цвет материала, убирая из него красный цвет и имея общее освещение, мы уже никак не сможем получить обратно черные кубы, мы можем получать более темные полутона. Таким образом, все определяется цветом освещения и выбранным материалом.

ИСПОЛЬЗОВАНИЕ БОЛЕЕ РЕАЛИСТИЧНОГО ОСВЕЩЕНИЯ

Можно предположить, что представленные таким образом кубы не выглядят очень реалистичными. Вы не можете даже видеть цент-

ральные вершины и грани куба, объект выглядит сплошным и нерельефным. Оказывается, все зависит от того, как общий свет освещает сцену. Если вы помните, при определении освещения отраженный свет вычислялся одинаково для всех вершин в сцене, независимо от нормали к плоскости объекта или любого другого параметра освещения, **рис.** 5.1.

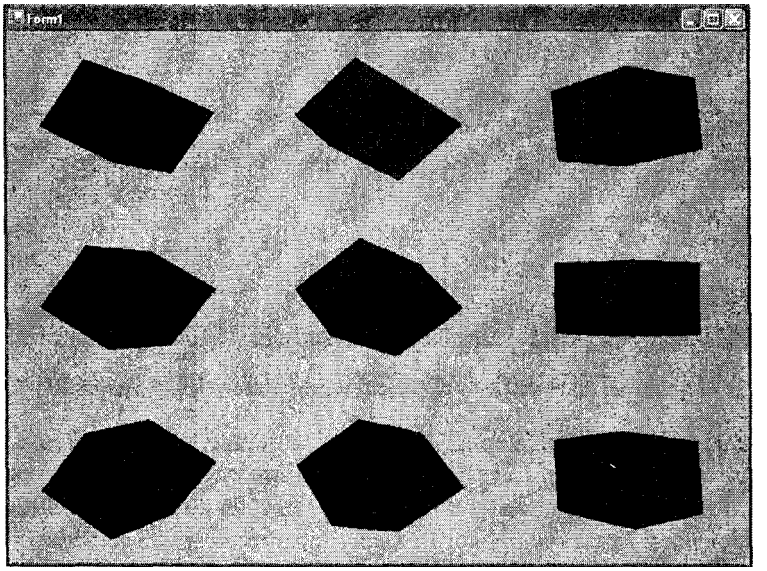


Рис. 5.1. Общее освещение без использования шейдинга (ретуширования)

Чтобы отобразить наш куб более реалистично, мы должны добавить реальный свет к нашей сцене. Просмотрите строку, описывающую общий свет, и после этого добавьте следующие строки:

```
device.Lights[0].Type = LightType.Directional;
device.Lights[0].Diffuse = Color.DarkBlue;
device.Lights[0].Direction = new Vector3(0, -1, -1);
device.Lights[0].Commit ();
device.Lights[0].Enabled = true;
```

Это создаст источник направленного темно-синего света, направление которого совпадает с выбранным направлением камеры. Теперь приложение отобразит затененные вращающиеся темно-синие кубы гораздо более реалистично. Можно увидеть, что направленные к наблюдателю грани освещены полностью, тогда как повернутые грани кажутся более затененными (возможно даже полностью темными), **рис.** 5.2.

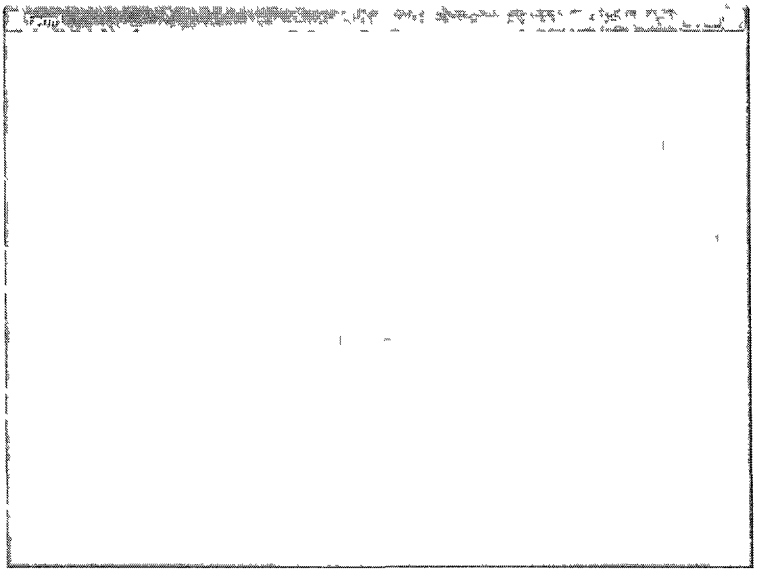


Рис. 5.2. Затененные кубы с направленным освещением

Имеются несколько готовых объектов, которые вы можете использовать при использовании Mesh-файлов. Используйте любой из следующих методов для создания этих объектов (каждая из готовых Mesh-функций требует в качестве первого параметра значение устройства):

mesh = Mesh.Box(device, 2.0f, 2.0f, 2.0f);

Метод, использующий левую систему координат для создания куба.

```
Width - ширина  Определяет размер куба вдоль оси X
Height - высота Определяет размер куба вдоль оси Y
Depth - глубина Определяет размер куба вдоль оси Z

mesh = Mesh.Cylinder(device, 2.0f, 2.0f, 2.0f, 36, 36);
```

Метод, использующий левую систему координат для создания цилиндра.

```
Radius1   Радиус цилиндра на отрицательном конце оси Z. Это значение
      должно быть больше или равно 0.0f.
Radius2   Радиус цилиндра на положительном конце оси Z. Это значение
      должно быть больше или равно 0.0f.
```

```
Length Длина цилиндра на оси Z.
Slices Число секторов (slices) вдоль главной оси (большее значение
       добавит больше вершин).
Stacks Число стеков вдоль главной оси (большее значение добавит
       больше вершин).

mesh = Mesh.Polygon(device, 2.0f, 8);
```

Метод, использующий левую систему координат для создания полигона.

```
Length Длина каждой стороны полигона.
Sides  Число сторон полигона.

mesh = Mesh.Sphere(device, 2.0f, 36, 36);
```

Метод, использующий левую систему координат для создания сферы.

```
Radius Радиус сферы. Это значение должно быть больше или равно 0
Slices Число секторов (slices) вдоль главной оси (большее значение
       добавит больше вершин).
Stacks Число стеков вдоль главной оси (большее значение добавит
       больше вершин).

mesh = Mesh.Torus(device, 0.5f, 2.0f, 36, 18);
```

Метод, использующий левую систему координат для создания тора.

```
InnerRadius  Внутренний радиус тора. Это значение должно быть больше или
       равно 0
OutterRadius Внешний радиус тора. Это значение должно быть больше или
       равно 0
Sides  Число сторон в поперечном сечении тора. Это значение должно
       быть больше или равно трем.
Rings  Число колец в поперечном сечении тора. Это значение должно
       быть больше или равно трем.

mesh = Mesh.Teapot(device) ;
```

Метод, использующий левую систему координат для создания изображения заварочного чайника , **рис.5.3.**

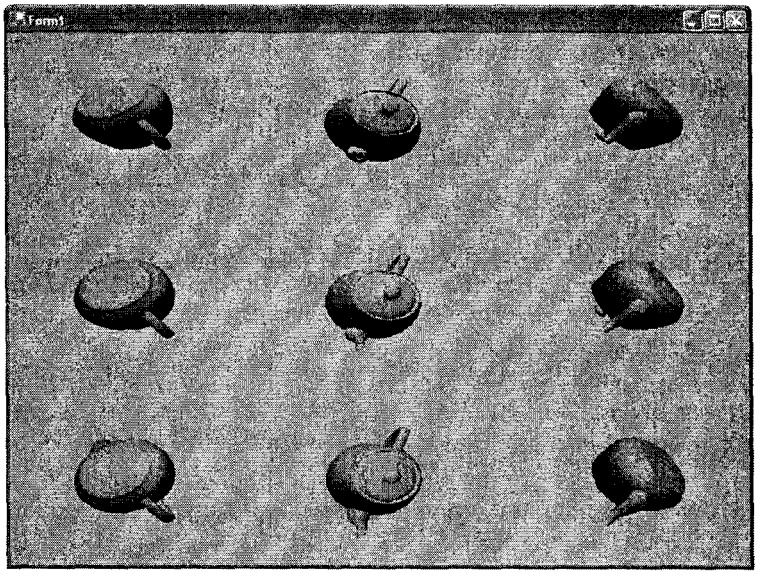


Рис. 5.3. Встроенный Mesh-обект в виде заварочного чайника

Каждый из описанных методов также имеет вторую перегрузку операций, которая может возвращать информацию о смежных вершинах в виде трех целых чисел на каждую сторону объекта, которые определяют три «соседних» элемента каждой стороны Mesh-объекта.

Использование Mesh-объектов для рендеринга сложных моделей

Отображение заварочных чайников выглядит вполне реалистично, но это не часто используется при программировании игр. Большинство Mesh-обьектов создано художниками, использующими приложения моделирования. Если ваше приложение моделирования поддерживает экспорт файлов формата .X, вам повезло!

ПРЕОБРАЗОВАНИЕ ОБЩИХ ФОРМАТОВ МОДЕЛИРОВАНИЯ В ФАЙЛ .X ФОРМАТА

DirectX SDK (включенный в CD диск) включает в себя несколько утилит преобразования для наиболее популярных приложений моделирования. Использование таких утилит позволяет вам легко преобразовывать, сохранять и использовать ваши высококачественные модели в ваших приложениях.

Существует несколько типов данных, сохраненных в обычном «.x» файле, который может быть загружен при создании Mesh-объектов. Сюда можно отнести вершинные и индексные данные, которые потребуются при выполнении модели. Каждое из Mesh-подмножеств будет иметь соответствующий материал. Набор материалов может также содержать информацию о текстуре. Вы можете также получить файл шейдера — высокоуровневого языка программирования для построения теней (дословно — High Level Shader Language, HLSL), используемого с этим Mesh-объектом при загрузке файла. Язык шейдеров HLSL — более совершенный раздел, который мы позже рассмотрим более подробно.

Помимо статических методов с использованием Mesh-объектов, которые позволили нам создавать наши «простые» типы примитивов, существуют еще два основных статических Mesh-метода, которые могут использоваться для загрузки внешних моделей. Они называются Mesh.FromFile (с использованием файла) и Mesh.FromStream (с использованием потока). Эти методы по существу идентичны, только в потоковом методе имеется большее количество перегрузок в зависимости от размера потока. Корневые перегрузки для каждого метода следующие:

```
public static Microsoft.DirectX.Direct3D.Mesh FromFile ( System.String
    filename ,
    Microsoft.DirectX.Direct3D.MeshFlags   options   ,
    Microsoft.DirectX.Direct3D.Device   device   ,
    Microsoft.DirectX.Direct3D.GraphicsStream   adjacency   ,
    out Microsoft.DirectX.Direct3D.ExtendedMaterial[] materials ,
    Microsoft.DirectX.Direct3D.EffectInstance   effects   )

public static Microsoft.DirectX.Direct3D.Mesh FromStream ( System.10.Stream
    stream ,
    System.Int32 readBytes ,
    Microsoft.DirectX.Direct3D.MeshFlags   options   ,
    Microsoft.DirectX.Direct3D.Device device ,
    Microsoft.DirectX.Direct3D.GraphicsStream   adjacency   ,
    out  Microsoft.DirectX.Direct3D.ExtendedMaterial[]   materials   ,
    Microsoft.DirectX.Direct3D.EffectInstance effects )
```

Первым параметром является тип источника данных *(filename* или *stream),* который мы будем использовать для загрузки Mesh-объекта. В случае загрузки из файла эта строка определяет имя загружаемого Mesh-файла. В случае использования потока строка указывает на поток и на число байтов, которые мы хотим считать для данных. При желании считать весь поток можно просто не включать значение *readBytes.*

Параметр MeshFlags управляет тем, где и как загружены данные. Этот параметр может быть представлен в виде поразрядной комбинации значений, см. таблицу 5.1.

Таблица 5.1. Значения параметра MeshFlags

Параметр	**Значение**
MeshFlags.DoNotClip	Использует флаг Usage.DoNotClip для вершинного и индексных буферов
MeshFlags.Dynamic	Равнозначное использование IbDynamic и VbDynamic
MeshFlags.IbDynamic	Использует Usage.Dynamic для индексных буферов
MeshFlags.IbManaged	Использует пул памяти Pool.Managed для индексных буферов
MeshFlags.IbSoftwareProcessing	Использует флаг Usage.SoftwareProcessing для индексных буферов
MeshFlags.IbSystemMem	Использует пул памяти Pool.SystemMemory для индексных буферов
MeshFlags.lbWriteOnly	Использует флаг Usage.WriteOnly Для индексных буферов
MeshFlags.VbDynamic	Использует Usage.Dynamic для вершинных буферов
MeshFlags. VbManaged	Использует пул памяти Pool.Managed для вершинных буферов
MeshFlags.VbSoftwareProcessing	Использует флаг Usage. SoftwareProcessing для вершинных буферов
MeshFlags. VbSystemMem	Использует пул памяти Pool.SystemMemory для вершинных буферов
MeshFlags. VbWriteOnly	Использует флаг Usage.WriteOnly для вершинных буферов
MeshFlags.Managed	Равнозначное использование IbManaged и VbManaged
MeshFlags.Npatches	Использование флага Usage.NPatches для индексных и вершинных буферов. При рендеринге Mesh объекта потребуется дополнительный улучшенный N-Patch
MeshFlags.Points	Использует флаг Usage.Points для индексных и вершинных буферов

Параметр	Значение
MeshFlags.RtPatches	Использует флаг Usage.RtPatches для индексных и вершинных буферов
MeshFlags.SoftwareProcessing	Равнозначное использование IbSoftwareProcessing и VbSoftwareProcessing
MeshFlags.SystemMemory	Равнозначное использование IbSystemMem и VbSystemMem
MeshFlags.Use32Bit	Использует 32-разрядные индексы для индексного буфера. Пока возможно, но обычно не рекомендуется
MeshFlags.UseHardwareOnly	Использует только аппаратную обработку

Следующий параметр *device* — устройство, которое мы будем использовать для рендеринга Mesh-объекта. Этот параметр обязательный, поскольку ресурсы должны быть связаны с устройством.

Параметр *adjacency* является внешним параметром и означает, что параметр будет локализован и попадет к вам после того, как функция закончит выполнение. Это возвратит информацию о смежных вершинах в виде трех целых чисел на поверхность объекта, которые определяют три «соседних» элемента каждой поверхности Mesh-объекта.

ЧТЕНИЕ ИНФОРМАЦИИ О СМЕЖНЫХ ЗНАЧЕНИЯХ ОТ ВОЗВРАЩАЕМОГО ПАРАМЕТРА *GRAPHICSSTREAM*

Информация о смежных значениях, возвращаемая вам процедурой создания Mesh-объекта, будет приходить из класса GraphicsStream. Вы можете получить локальную копию смежных данных с помощью следующего кода:

```
int[] adjency = adjBuffer.Read(typeaf(int),
mesh.NumberFaces * 3);
```

Это приведет к созданию массива из трех целых чисел на каждую поверхность объекта, в котором сохранится информация о смежных данных, и к которой проще обращаться, чем непосредственно к классу GraphicsStream.

Параметр расширения *materials* также является выходным параметром, который возвратит массив, включающий информацию о различных

подмножествах Mesh. Класс ExtendedMaterial поддерживает как обычный Direct3D материал, так и строку «string», которая может использоваться для загрузки текстур. Обычно этой строкой является имя файла или название ресурса текстуры. Однако, поскольку загрузка текстуры выполняется в виде приложения, это могут быть любые введенные пользователем строковые данные.

Наконец, последний параметр *EffectInstance* описывает файл шейдера HLSL и значения, которые будут использоваться для данного Mesh-объекта. Существуют множественные перегрузки для каждого из этих методов, которые используют различные комбинации этих параметров. Старайтесь выбирать тот метод, который имеет только необходимую информацию.

Теперь, проанализировав необходимую информацию, мы готовы к тому, чтобы загружать и отображать Mesh-объекты. Это выглядит слегка пугающе вначале, но на самом деле, не представляет больших сложностей. Попробуем начать писать соответствующий код.

Сначала мы должны удостовериться, что мы имеем переменные объекта, которые позволят сохранить наши материалы и текстуры для различных подмножеств Mesh. Добавьте следующие переменные после объявления Mesh-объекта:

```
private Material[] meshMaterials;
private Texture[] meshTextures;
```

Поскольку встречается много различных подмножеств объектов, необходимо сохранить как массив текстур, так и массив материалов, по одному для каждого подмножества. Давайте рассмотрим некоторый код, позволяющий загрузить Mesh-объект (функция LoadMesh), см. листинг 5.1:

```
Листинг 5.1. Загрузка Mesh объекта из файла.

private void LoadMesh(string file)
{
   ExtendedMaterial[] mtrl;
   // Load our mesh
   mesh = Mesh.FromFile(file, MeshFlags.Managed, device, out mtrl);
   // If we have any materials, store them
   if ((mtrl != null) && (mtrl.Length > 0))
   {
      meshMaterials = new Material[mtrl.Length];
      meshTextures = new Texture[mtrl.Length];
      // Store each material and texture
      for (int i = 0; i < mtrl.Length; i++)
      {
```

```
            meshMaterials[i] = mtrl[i].Material3D;
            if ((mtrl[i].TextureFilename != null) && (mtrl[i].TextureFilename !=
                string.Empty))
            {
                // We have a texture, try to load it
                meshTextures[i] = TextureLoader.FromFile(device, @"..\..\" +
                    mtrl[i].TextureFilename);
            }
        }
    }
```

Сначала мы объявляем наш массив ExtendedMaterial, в котором будет храниться информация о подмножествах Mesh-объекта. Затем мы просто вызываем метод FromFile для загрузки объекта. На данном этапе мы не учитываем смежные или HLSL-параметры, поэтому используем перегрузку без них.

После загрузки Mesh-объекта необходимо сохранить информацию о материале и текстурах для различных подмножеств. После того как мы удостоверимся в наличии последних, мы, наконец, можем распределить массивы материалов и текстур, использующих число подмножеств в качестве размера. Затем мы свяжем каждое из наших значений массива ExtendedMaterial и сохраним материал в нашей локальной копии. Если имеется информация о текстуре, включенной в это подмножество материалов, то для создания текстуры мы используем функцию TextureLoader.FromFile. Эта функция определена только двумя параметрами, устройством и именем файла текстуры, и является более предпочтительной, чем выполнение с помощью функции System.Drawing.Bitmap, рассмотренной раньше.

Для рисования данного Mesh-объекта добавьте следующий метод к нашему приложению:

```
private void DrawMesh(float yaw, float pitch, float roll, float x, float y,
float z)
 {
    angle += 0.01f;
    device.Transform.World = Matrix.RotationYawPitchRoll(yaw, pitch, roll) *
    Matrix.Translation(x, y, z);
    for (int i = 0; i < meshMaterials.Length; i++)
    {
        device.Material = meshMaterials[i];
        device.SetTexture(0, meshTextures[i]);
        mesh.DrawSubset(i);
```

Если вы обратили внимание, мы сохранили ту же самую последовательность написания кода, что и при использовании метода DrawBox. Далее, чтобы рисовать Mesh-объект, необходимо связать все материалы и выполнить следующее:

1. Установить сохраненный материал как материал для устройства.
2. Установить значение текстуры в устройстве для сохраненной текстуры. Даже если не имеется никакой сохраненной текстуры, установите значением текстуры нулевое значение или пустой указатель.
3. Вызвать функцию пересылки DrawSubset в нашем идентификаторе подмножества.

Замечательно, теперь мы имеем все, что необходимо для загрузки Mesh-объекта и отображения его на экране. Исходный текст нашего Mesh-объекта «tiny.x» находится в прилагаемом CD диске, данный файл является тестовым для приложения DirectX SDK. Чтобы его запустить мы добавили следующие строки после создания устройства:

```
// Load our mesh
LoadMesh(@"..\..\tiny.x");
```

Затем мы должны изменить нашу информацию о камере, поскольку загружаемый объект оказался достаточно большим. То есть, мы должны изменить масштаб изображения нашей камеры. Замените представление и преобразование проекции следующим образом:

```
device.Transform.Projection = Matrix.PerspectiveFovLH((float)Math.PI / 4,
   this.Width /
   this.Height, 1.0f, 10000.0f);
device.Transform.View = Matrix.LookAtLH(new Vector3(0,0, 580.0f), new
   Vector3(),
   new Vector3(0,1,0));
```

Как видно, мы увеличили длину задней плоскости и переместили камеру достаточно далеко назад. Осталось вызвать функцию DrawMesh:

```
DrawMesh(angle / (float)Math.PI, angle / (float)Math.PI * 2.0f, angle /
   (float)Math.PI / 4.0f, 0.0f, 0.0f, 0.0f);
```

Запуск и выполнение приложения отобразит на экране картинку, приведенную на **рис.**5.4.

Рис. 5.4 Рендеринг Mesh-объекта, загруженного из файла

Итак, мы получили нечто более реалистичное, чем просто вращающиеся кубы или треугольники.

Краткие выводы

В этой главе мы охватили вопросы рендеринга с использованием Mesh-объектов, включая.

- Использование объектов Mesh
- Использование материалов.
- Использование Mesh для рисования общих объектов.
- Использование Mesh для загрузки и отображения внешних файлов.

В нашей следующей главе мы рассмотрим вопросы оптимизации данных Mesh-объектов и ознакомимся с более совершенными особенностями этой технологии.

Глава 6. Использование Управляемого DirectX для программирования игр

Выбор игры

Несмотря на то, что мы не обсудили более совершенные возможности трехмерного программирования, у нас имеется достаточно информации для того, чтобы написать простую трехмерную игру.

Прежде чем начать писать игру, было бы неплохо придумать план. Мы должны знать тип игры, которую мы будем писать, основные особенности и пр. Учитывая ограниченный набор разделов, которые мы охватили, мы не можем пока создавать что-то чрезмерно сложное. Вместо этого, мы напишем достаточно несложную программу для нашей игры.

Одна из первых демонстрационных игр для MS DOS называлась «Ослик», «Donkey», в ней пользователь управлял автомобилем, и цель игры состояла в том, чтобы объехать осликов, встречающихся на дороге. Игра или ее оформление представляются достаточно простыми для того, чтобы воссоздать ее на данный момент. В данной главе будет рассмотрено программирование этой игры в трехмерной варианте, но без осликов. Мы назовем эту игру «Dodger».

Итак, потратим немного времени на планирование и проектирование нашей игры. Что нам необходимо, и что мы хотим сделать? Очевидно, нам понадобится класс «Автомобиль» или «Car», чтобы управлять нашим транспортным средством. Затем, было бы неплохо иметь класс, отвечающий за управление препятствиями, которые мы будем пробовать объезжать. Плюс, мы будем нуждаться в нашем основном классе — движке игры, который будет выполнять весь рендеринг и связывать все это вместе.

Если вы попытаетесь создать коммерческий вариант игры, вам понадобится затратить много времени на разработку описания игры, что является достаточно серьезным документом, детализирующим огромное количество игровых концепций и особенностей. Основная задача этой книги состоит в том, чтобы охватить процесс программирования игр, а не шаги, которые должны привести к изданию, поэтому вопросы коммерциализации мы опустим.

Другой рекомендуемый документ, который обычно требуется перед началом работы по разработке — техническая спецификация (для краткости, спецификация). Она включает детальный список всех классов, а также методы и свойства, которые эти классы осуществят. Этот документ может также содержать диаграммы UML, которые отображают отношения между объектами.

Цель спецификации состоит в том, чтобы изначально и основательно обдумать проект вашего приложения, прежде чем вы начнете писать какой-либо код или программу. Поскольку акцент в книге поставлен на программирование игр, непосредственно вопросы по спецификации данного проекта мы также опустим. Но на будущее рекомендуется всегда продумывать спецификацию перед написанием любого кода.

Программирование игры

Вы можете запустить приложение Visual Studio и создать новый проект. Создайте новый проект C# Windows Application под именем «Dodger». Следует отметить, что заданное по умолчанию название формы — Form1. Замените каждое имя Form1 на имя DodgerGame, которое будет являться названием класса и будет представлено соответствующим кодом в этой главе. Необходимо добавить ссылки на три сборки Управляемого DirectX, которые мы уже использовали в проектах раньше, и включить для них using-директиву. Перепишите конструктор следующим образом:

```
public DodgerGame()
(
   this.Size = new Size(800,600);
   this.Text = "Dodger Game";
   this.SetStyle(ControlStyles.AllPaintinglnWmPaint | ControlStyles.Opaque,
   true);
}
```

Данная процедура установит размер окна 800x600, заголовок окна и стиль для корректного выполнения рендеринга. Затем необходимо изменить точку входа, заменяя основной метод «Main» на приведенный в листинге 6.1.

Листинг 6.1. Основная точка входа игры.

```
static void Main()
{
   using (DodgerGame frm = new DodgerGame!))
   {
      // Show our form and initialize our graphics engine
      frm.Show();
      frm.InitializeGraphics();
      Application.Run(frm);
   }
}
```

По существу, это тот же самый код, с помощью которого мы уже запускали все примеры раньше. Мы создаем окно Windows, инициализируем графику и затем запускаем форму, и таким образом, приложение. Тем не менее, в функции инициализации InitializeGraphics, где и будет запускаться программа, необходимо внести некоторые изменения. Добавьте к вашему приложению метод, приведенный в листинге 6.2.

Листинг 6.2. Инициализация Графических Компонентов.

```
/// <summary>
/// We will initialize our graphics device here
/// </summary>
public void InitializeGraphics!)
(
   // Set our presentation parameters
   PresentParameters presentParams = new PresentParametersO;
   presentParams.Windowed = true;
   presentParams.SwapEffeet = SwapEffect.Discard;
   presentParams.AutoDepthStencilFormat = DepthFormat.D16;
   presentParams.EnableAutoDepthStencil = true;
   // Store the default adapter
   int adapterOrdinal = Manager.Adapters.Default.Adapter;
   CreateFlags flags = CreateFlags.SoftwareVertexProcessing;
   // Check to see if we can use a pure hardware device
   Caps caps = Manager.GetDeviceCaps(adapterOrdinal, DeviceType.Hardware);
   // Do we support hardware vertex processing?
   if (caps.DeviceCaps.SupportsHardwareTransformAndLight)
      // Replace the software vertex processing
      flags = CreateFlags.HardwareVertexProcessing;
   // Do we support a pure device?
   if (caps.DeviceCaps.SupportsPureDevice)
      flags ] = CreateFlags.PureDevice;
   // Create our device
   device = new Device(adapterOrdinal, DeviceType.Hardware, this, flags,
   presentParams);
   // Hook the device reset event
   device.DeviceReset += new EventHandler(this.OnDeviceReset);
   this.OnDeviceReset(device, null);
```

Сначала создается структура параметров представления, и аналогично тому, как мы это уже делали, определяется буфер глубины. Затем сохраняется порядковый номер адаптера (заданный по умолчанию) и флажки создания устройств, после чего по умолчанию устанавливаются параметры обработки вершин.

На сегодняшний день современные графические платы могут поддерживать обработку вершин за счет аппаратного ускорения. Зачем же тратить процессорное время, когда можно «возложить» часть операций непосредственно на графическую карту, которая выполнит это значительно быстрее? Однако, вы можете не знать, действительно ли используемый адаптер поддерживает эти возможности. Об этом немного позже.

Теперь мы можем узнавать о возможностях устройства еще до его создания, а также определять флажки, необходимые при создании устройства. Как вы помните из главы 2, структура отображаемого списка возможностей устройства огромна и разбита на различные подразделы. Подраздел, представляющий интерес на данный момент, — DeviceCaps, содержит описание свойств и возможностей для соответствующего драйвера.

Когда вы хотите выяснить, поддерживается ли данная специфическая возможность или нет, вы можете просто проверить булево значение, относящееся к этой возможности: если это значение «true», возможность поддерживается, в противном случае, нет. В первую очередь вы выясняете, поддерживаются ли в данном устройстве аппаратные преобразования и освещение. Если да, вы можете создавать устройство с аппаратной обработкой вершин, добавив флаг hardware vertex processing к остальным. Затем необходимо выяснить, можете ли вы создавать реальное устройство (возможно только при наличии аппаратной обработки вершин); если да, вы используете для флажков поразрядный оператор «OR» («или»), добавляя также эту возможность. Реальное аппаратное устройство — наиболее эффективный тип устройства, которое вы можете создать, так что если эти опции доступны, необходимо использовать их.

Затем вы создаете устройство, используя указанные флажки, в зависимости от возможностей ваших графических плат, вы можете использовать реальное аппаратное устройство или же некую его разновидность. Если помните, при использовании вершинных буферов возникала необходимость отслеживания события сброса устройства, подобная же ситуация присутствует и здесь. Всякий раз, когда устройство сбрасывается, желательно определить установки состояния устройства по умолчанию. Поэтому необходимо отслеживать событие сброса. Метод обработчика событий или обработчик событий (event handler method) приведен в листинге 6.3, добавьте его к вашему приложению.

Листинг 6.3. Обработчик события сброса устройства

```
private void OnDeviceReset(object sender, EventArgs e)
(
   device.Transform.Projection = Matrix.PerspectiveFovLH((float)Math.PI / 4,
      this.Width / this.Height, 1.0f, 1000.0f);
```

```
device.Transform.View = Matrix.LookAtLH(new Vector3(0.0f, 9.5f, 17.Of),
   new Vector3(),
   new Vector3 (0,1,0));
// Do we have enough support for lights?
if  ((device.DeviceCaps.VertexProcessingCaps.SupportsDirectionalLights) &&
    ((unit)device.DeviceCaps.MaxActiveLights > 1))
(
   // First light
   device.Lights[0].Type = LightType.Directional;
   device.Lights[0].Diffuse = Color.White;
   device.Lights[0].Direction = new Vector3(l, -1, -1);
   device.Lights[0]. Commit ();
   device.Lights[0].Enabled = true;
   // Second light
   device.Lights[1].Type = LightType.Directional;
   device.Lights[l].Diffuse = Color.White;
   device.Lights[1].Direction = new Vector3(-l, 1, -1);
   device. Lights [ 1 ]. Commit () ;
   device.Lights[l].Enabled = true;
1
else
(
   // Hmm.. no light support, let's just use
   // ambient light
   device.RenderState.Ambient = Color.White;
1
```

Начало этой функции достаточно знакомо для нас. Устанавливается камера, определяется вид и преобразование проекции в устройстве. Для этой игры выбирается неподвижная камера, и эти параметры необходимо устанавливать после каждого сброса устройства (все состояния устройства аннулируются при сбросе).

Использование общего освещения не является предпочтительным, поскольку мы уже видели, что общий свет не совсем реалистичен, более подходящим был бы направленный свет. Однако, мы уже не можем проверить, поддерживает ли устройство данный тип освещения или нет, поскольку после создания устройства структура Properties, «возможности» или «свойства», уже не используется. Если устройство может поддерживать источник или источники направленного освещения, мы будем использовать их, в противном случае возможна установка общего света по умолчанию. Возможно, это будет не так реалистично, но это лучше, чем иметь неосвещенную сцену.

Необходимо окончательно переписать метод OnPaint для того, чтобы запустить процедуру рендеринга. Добавьте следующую функцию:

```
protected override void OnPaint(System.Windows.Forms.PaintEventArgs e)
{
   device.Clear(ClearFlags.Target \ ClearFlags.ZBuffer, Color.Black, 1.Of, 0);
   device.BeginScene();
   device.EndScene();
   device.Present();
   this.Invalidated ;
}
```

ИСПОЛЬЗОВАНИЕ ТОЛЬКО НЕОБХОДИМОГО ОСВЕЩЕНИЯ

Вместо того чтобы использовать подход «все или ничего» в плане выбора типа освещения, вы можете просто проводить «многоуровневую» проверку поддержки режимов освещения. В данном сценарии сначала выясняется, поддерживается ли хотя бы один источник света, и если да, можно включить его. Затем проверяется наличие второго поддерживаемого источника света, и так далее. Это дает возможность иметь резервный вариант (один источник) даже для тех устройств, которые не поддерживают два источника света и более. Данный пример не совсем подходит при использовании одного источника направленного света, поскольку многоуровневая проверка не была выполнена. Хотя даже в этом случае это выглядело бы примерно так:

```
// Do we have enough support for lights?
if ((device.DeviceCaps.VertexProcessingCaps.SupportsDirectionalLights) &&
   ((unit)device.DeviceCaps.MaxActiveLights > 0))
(
   II First light
   device.Lights[0].Type = LightType.Directional;
   device.Lights[0].Diffuse = Color.White;
   device.Lights[0].Direction = new
Vector3(l, -1, -1);
   device.Lights[0].Commit();
   device.Lights[0].Enabled = true;
   if ((unit)device.DeviceCaps.MaxActiveLights > 1))
   {
   // Second light
      device.Lights[l].Type = LightType. Directional;
      device.Lights[l].Diffuse = Color.White;
      device.Lights[1].Direction = new Vector3(-1, 1, -1);
      device.Lights[1].Commit();
      device.Lights[1].Enabled = true;
      }
      }
```

Вернемся к методу OnPaint. Ничего нового, кроме установки черного цвета фона. Теперь мы можем приступить к созданию первого игрового объекта — «road». Исходный текст программы находится на CD диске, включая файл .X, который будет отображать данные объекта «road», поэтому мы должны объявить переменные для объекта «road mesh»:

```
// Game board mesh information
private Mesh roadMesh = null;
private Material[] roadMaterials = null;
private Texture[] roadTextures = null;
```

Мы будем также использовать разновидности функции загрузки Mesh-объекта, которую мы описали в предыдущей главе. Главные различия здесь состоят в том, что это будет статический объект, поэтому потребуется вызывать данные из более чем одного класса и просматривать информацию о всех материалах и текстурах, в отличии от переменных класса level, которые использовались прежде. Добавьте следующий текст к вашему коду, см. листинг 6.4.

Листинг 6.4. Процедура загрузки Mesh-объекта.

```
public static Mesh LoadMesh(Device device, string file, ref Material!]
   meshMaterials,
   ref Texture[] meshTextures)
 {
   ExtendedMaterial[] mtrl;
   // Load our mesh
   Mesh tempMesh = Mesh.FromFile(file, MeshFlags.Managed, device, out mtrl);
   // If we have any materials, store them
   if ((mtrl != null) && (mtrl.Length > 0))
    {
      meshMaterials = new Material[mtrl.Length];
      meshTextures = new Texture[mtrl.Length];
      // Store each material and texture
      for (int i = 0; i < mtrl.Length; i++)
      (
         meshMaterials [i] = mtrl[i].Material3D;
         if ((mtrl[i].TextureFilename != null) && (mtrl[i].TextureFilename ! =
            string.Empty))
          (
            // We have a texture, try to load it
            meshTextures[i] = TextureLoader.FromFile(device, @"..\..\" +
            mtrl[i].TextureFilename);
      }
    }
```

```
    return tempMesh;
}
```

Эта функция уже обсуждалась раньше, так что нет необходимости описывать ее еще раз. Мы используем эту функцию, чтобы загрузить обьект «road», и нам следует определить и вызвать ее в процедуре обработчика события сброса устройства, в конце которой можно добавить следующий код:

```
// Create our road mesh
roadMesh = LoadMesh(device, @"..\..\road.x", ref roadMaterials, ref
roadTextures) ;
```

Удостоверьтесь в том, что вы скопировали объект файл текстуры в каталог с вашим исходным кодом. Этот код загрузит объект Mesh «road», включая текстуры, и сохранит текстуры, материалы и объект. Теперь, когда вам нужно отобразить на экране объект дороги больше чем один раз в кадре, необходимо создать функцию рендеринга. Добавьте следующую функцию к вашему коду:

```
private void DrawRoad(float x, float y, float z)
(
   device.Transform.World = Matrix.Translation(x, y, z);
   for (int i = 0; i < roadMaterials.Length; i++)
   (
      device.Material = roadMaterials[i];
      device.SetTexture (0, roadTextures [i]) ;
      roadMesh.DrawSubset(i);
```

Функция достаточно знакомая, мы использовали ее при рендеринге простых Mesh-объектов. Объект перемещается в соответствующее местоположение, и отображается каждое подмножество. План рисования нашей дороги состоит в том, чтобы отобразить последовательные перемещения автомобиля (два этапа: в текущий момент и сразу после этого). В действительности, автомобиль не будет перемещаться вообще, вместо этого мы будем перемещать дорогу.

Причина, требующая такого подхода, двоякая. Первое, если бы автомобиль перемещался в каждом отдельном кадре, нам следовало бы перемещать камеру, также привязываясь к кадру, чтобы не отставать от объекта. Это лишние вычисления, которые нам не нужны. Другая причина — точность: если бы мы позволили автомобилю продвигаться только вперед, и игрок был бы достаточно резвым, в конечном счете, автомобиль «уехал»

бы очень далеко в пространстве мировых координат, что привело бы к потере значащих цифр или даже переполнению переменной. Поскольку пространство перемещений не ограничено, мы оставляем автомобиль в том же самом положении и перемещаем только дорогу сверху вниз.

Соответственно, нам необходимо добавить некоторые переменные, чтобы управлять дорогой. Добавьте следующие переменные класса level и константы:

```
// Constant values for the locations
public const float RoadLocationLeft = 2.5f;
public const float RoadLocationRight = -2.5f;
private const float RoadSize = 100.0f;
private const float MaximumRoadSpeed = 250.0f;
private const float RoadSpeedIncrement = 0.5f;
// Depth locations of the two 'road' meshes we will draw
private float RoadDepth0 = 0.0f;
private float RoadDepth1 = -100.0f;
private float RoadSpeed = 30.0f;
```

Mesh-объект «road», используемый для создания дороги — достаточно распространенный объект. Его длина составляет 100 единиц, а ширина — 10 единиц. Константа размера отражает фактическую длину дороги, в то время как две константы местоположения отмечают расстояние от центральной линии линий обеих сторон дороги. Последние две константы предназначены для управления в процессе ведения игры. Максимальная скорость перемещения дороги составляет 250 единиц в секунду, при увеличении скорости дискретность составляет половину единицы.

Наконец, необходимо установить значение глубины двух дорожных секций. Для этого следует инициализировать первую секцию как ноль, а вторую секцию начать непосредственно с конца первой (обратите внимание, что это значение равно размеру дороги). Итак, мы имеем основные переменные и константы, необходимые для рисования и перемещения дороги, и можно добавить вызов процедуры рисования. Поскольку мы хотим отобразить дорогу первой, добавьте два вызова DrawRoad после функции рендеринга сцены BeginScene:

```
// Draw the two cycling roads
DrawRoad(0.0f, 0.0f, RoadDepth0);
DrawRoad(0.0f, 0.0f, RoadDepth1);
```

Запустив приложение, видим, что дорога отображается на экране, однако асфальт дороги смотрится чрезвычайно пикселизованным, несплошным. Причиной такой пикселизации является способ, через который Direct3D определяет цвет пиксела в представленной сцене. Когда один элемент текстуры — тексел — охватывает больше чем один пиксел на

экране, пикселы рассчитываются фильтром растяжения. Когда несколько элементов текстуры перекрывают отдельный пиксел, они рассчитываются фильтром сжатия. Заданный по умолчанию фильтр растяжения и сжатия, называемый точечным фильтром (Point Filter), попросту использует самый близкий элемент текстуры как цвет для соответствующего пиксела. Это и вызывает эффект пикселизации.

Существуют различные способы для фильтрации текстур, однако, не каждое устройство может поддерживать их. Все, что вам действительно необходимо, это фильтр, который может интерполировать элементы текстуры дороги, чтобы выполнить ее более гладко. К функции OnDeviceReset добавьте код, приведенный в листинге 6.5.

Листинг 6.5. Фильтрация текстуры.

```
// Try to set up a texture minify filter, pick anisotropic first
if  (device.DeviceCaps.TextureFilterCaps.SupportsMinifyAnisotropic)
{
   device.SamplerState[0].MinFilter  =  TextureFilter.Anisotropic;
}
else  if  (device.DeviceCaps.TextureFilterCaps.SupportsMinifyLinear)
{
   device.SamplerState[0].MinFilter  =  TextureFilter.Linear;
}
// Do the same thing for magnify filter
if  (device.DeviceCaps.TextureFilterCaps.SupportsMagnifyAnisotropic)
{
   device.SamplerState[0].MagFilter  =  TextureFilter.Anisotropic;
}
else  if  (device.DeviceCaps.TextureFilterCaps.SupportsMagnifyLinear)
{
   device.SamplerState[0].MagFilter  =  TextureFilter.Linear;
}
```

Как вы можете здесь видеть, вначале выясняется, способно ли ваше устройство поддерживать анизотропную фильтрацию при растяжении или сжатии изображения. Если да, то вы можете использовать этот фильтр хтя обоих режимов. Если нет, то далее выясняется, поддерживает ли устройство линейную фильтрацию для этих же режимов. Если ни один из фильтров не поддерживается, вы ничего не сможете сделать с пикселизацией дороги. В случае если плата поддерживает один из этих методов фильтрации, приложение отобразит на экране более гладкую поверхность дороги.

Теперь дорога находится в середине экрана, но еще не перемещается. Понадобится новый метод, использующийся для обновления состояния игры, который позволит выполнить перемещение дороги и отследить

столкновение автомобиля с препятствием. Для этого необходимо вызвать эту функцию в разделе метода OnPaint (до вызова функции очистки Clear):

```
// Before this render, we should update any state
OnFrameUpdate ();
```

Также необходимо добавить к приложению метод, приведенный в листинге 6.6.

Листинг 6.6. Метод обновления кадра.

```
private void OnFrameUpdate ()
 {
    // First, get the elapsed time
    elapsedTime = Utility.TimerjDirectXTimer.GetElapsedlime);
    RoadDepth0 += (RoadSpeed * elapsedTime);
    RoadDepth1 += (RoadSpeed * elapsedTime);
    // Check to see if we need to cycle the road
    if (RoadDepth0 > 75.0f)
    {

       RoadDepth0 = RoadDepth1 - 100.0f;

       }

    if (RoadDepth1 > 75.0f)

    {

       RoadDepth1 = RoadDepth0 - 100.0f;
 }
```

Данная программа будет содержать гораздо больше строк, чем сейчас, прежде чем написание игры будет завершено, но пока все, что нам действительно необходимо, это — перемещение дороги. Игнорирование параметра *elapsed time* позволяет перемещать дорогу и затем удалять пройденные дорожные секции, размещая их в конце текущей дорожной секции. При этом необходимо определить «количество» пройденной дороги, умножив текущую дорожную скорость (измеряемую в единицах за секунду) на количество прошедшего времени (в секундах), таким образом мы получаем «количество» дороги на кадр. Также необходимо включить ссылку на *elapsedTime* в вашей секции объявления переменных:

```
private float elapsedTime = 0.0f;
```

ПЕРЕМЕЩЕНИЕ ОБЪЕКТОВ В РЕАЛЬНОМ МАСШТАБЕ ВРЕМЕНИ

Почему это так необходимо? Скажем, вы решили увеличивать приращение величины дороги при постоянном значении для каждого кадра. На вашем компьютере это выполняется совершенно, так почему это не работает точно также на других системах? Например, дорога перемещается на другом компьютере несколько медленнее, чем на вашем. Или дорога перемещается удивительно медленно, по сравнению с тем, что испытывает человек, едущий на машине.

Причина кроется в том, что вы выполняете ваши вычисления, опираясь на частоту смены кадров. Например, скажем, в вашей системе изображение сменяется со скоростью 60 кадров в секунду, и все вычисления опираются на это значение. Теперь возьмем машины, которые работают с частотой обновления кадра 40 кадров в секунду, или более быстрые, например, 80 кадров в секунду; естественно мы получим различные результаты. Таким образом, возникает задача, чтобы ваша игра выполнялась неизменно и независимо от типа системы, поэтому при вычислениях нужно уходить от привязки к частоте смены кадров.

Лучший способ для решения этой проблемы состоит в том, чтобы определить и привязать игровые перемещения и вычисления к некоторой неизменной единице времени. Например, максимальная скорость нашей дороги определена как 250 единиц в секунду. Наша первая цель состояла бы в том, чтобы определить время, прошедшее с момента нашего последнего обновления. В версии .NET Runtime имеется встроенная утилита (встроенный таймер), которая может использоваться для определения текущего отсчета времени системы, но у которой есть свой недостаток, связанный главным образом с низкой разрешающей способностью таймера, составляющей, как правило, 15 миллисекунд. Это приводит к тому, что при высокой скорости смены кадров (более 60 кадров в секунду) движения будут казаться прерывистыми.

Версия DirectX SDK включает в себя класс, называемый DirectXTimer, который, если ваша машина позволяет это, использует таймер с высоким разрешением (обычно 1 миллисекунда). Если данный таймер не доступен на вашей машине, то система вернется к встроенному таймеру. Примеры в этой книге будут использовать данный таймер (DirectXTimer) как механизм определения времени. Он уже включает в себя код для таймера высокой точности, поэтому мы не будем изобретать это «колесо» дважды.

Добавление движущегося автомобиля в используемую сцену

Теперь, когда вы отображаете на экране объект в виде перемещающейся дороги, необходимо добавить объект, с которым взаимодействует игрок — это автомобиль (в листинге программы — «car»). Вы могли бы просто разместить переменные класса «car», константы и код в основной класс — «Main class», в котором размещены и дорожные секции, но это выглядело бы не так модульно, как хотелось. Было бы предпочтительнее размещать автомобильный код отдельно, в его собственном классе, который мы попробуем сейчас создать. В главном меню проекта откройте опцию Add New Item и выберите в качестве добавляемого новый класс, названный «car». Проверьте, что вы включили операторы использования Управляемого DirectX в этом новом файле кода, созданном для вашего приложения.

Что же должен делать класс «car»? Как мы уже говорили, в процессе рендеринга автомобиль остается неподвижным относительно перемещающейся сцены, за исключением перемещений влево-вправо, когда он должен объезжать препятствия. С учетом вышесказанного, необходимо добавить следующий набор переменных в наш класс:

```
// Car constants
public const float Height = 2.5f;
public const float Depth = 3.0f;
public const float SpeedIncrement = 0.If;
private const float Scale = 0.85f;
// Car data information
private float carLocation = DodgerGame.RoadLocationLeft;
private float carDiameter;
private float carSpeed = 10.Of;
private bool movingLeft = false;
private bool movingRight = false;
// Our car mesh information
private Mesh carMesh = null;
private Material[] carMaterials = null;
private Texture[] carlextures = null;
```

Данные переменные будут управлять всеми параметрами, необходимыми для управления автомобилем. Константы высоты «Height» и глубины «Depth» автомобиля останутся статическими (поскольку перемещение осуществляется только влево-вправо). Приращение скорости бокового перемещения «SpeedIncrement» также будет постоянным. Последняя константа «Scale» — масштаб или отношение размера автомобиля относительно ширины дороги.

Переменные в классе «car» очевидны. Текущее местоположение автомобиля на дороге, которое установлено по умолчанию с левой стороны дороги. Диаметр автомобиля, который будет использоваться при столкновении автомобиля с препятствием. Текущая боковая скорость автомобиля (так как скорость перемещения дороги может увеличиваться, скорость перемещения автомобиля должна увеличиться соответственно). И, наконец, две логические переменные, которые определяют направление перемещения (влево-вправо), а также файл .X данных объекта Mesh.

Для создания объекта Mesh (и его связанных структур) и вычисления диаметра автомобиля потребуется конструктор класса car. Замените заданный по умолчанию конструктор на специально созданный для вашего приложения, см. листинг 6.7.

Листинг 6.7. Создание класса car

```
/// <summary>
/// Create a new car device, and load the mesh data for it
/// </summary>
/// <param name="device">D3D device to use</param>
public Car(Device device)
{
   // Create our car mesh
   carMesh = DodgerGame.LoadMesh(device, @"..\..\car.x", ref carMaterials,
ref carTextures);
   // We need to calculate a bounding sphere for our car
   VertexBuffer vb = carMesh.VertexBuffer;
   try
   {
      // We need to lock the entire buffer to calculate this
      GraphicsStream stm = vb.Lock(0, 0, LockFlags.None);
      Vector3 center; // We won't use the center, but it's required
      float radius = Geometry.ComputeBoundingSphere(stm,
         carMesh.NumberVertices, carMesh.VertexFormat, out center);
      // All we care about is the diameter. Store that
      carDiameter = (radius * 2) * Scale;
   }
   finally
   {
      // No matter what, make sure we unlock and dispose this vertex
      // buffer.
      vb.Unlock();
      vb.Disposed;
   }
}
```

Создание объекта Mesh достаточно просто, так как представляет собой тот же самый метод, который мы использовали при создании Mesh-объекта road, только с другими именами переменных. Новым алгоритмом здесь является только вычисление диаметра автомобиля. Вычисляется граничная сфера (сфера, которая полностью заключает в себе все точки в Mesh объекте) для автомобиля. Класс geometry содержит данную функцию, но она требует пересылки параметров вершин для вычисления граничной сферы.

То, что нам сейчас необходимо, это получить вершинные данные из Mesh-объекта. Уже известно, что данные вершин хранятся в вершинных буферах, так что мы можем использовать вершинный буфер, сохраненный в Mesh-объекте. Чтобы считать данные из вершинного буфера необходимо вызвать метод блокировки на момент считывания, Lock method, который возвращает все потоковые вершинные данные. О других методах, работающих с вершинными буферами, мы расскажем в следующей главе. После метода блокировки мы можем использовать метод ComputeBoundingSphere, чтобы получить «центр» этого Mesh-объекта и радиус сферы. Поскольку мы не заботимся о центре объекта и хотим сохранить диаметр, удвоим радиус и сохраним его. Окончательно (в блоке finally), мы проверяем, что вершинный буфер, который мы использовали, разблокирован и освобожден.

Далее необходимо добавить метод, выполняющий прорисовку вашего автомобиля. Поскольку вы сохраняете положение автомобиля в классе, единственная необходимая вещь, которая понадобится для этого метода — это используемое для рисунка устройство. Этот метод аналогичен методу DrawRoad, отличаясь лишь в использовании переменных и в том, что мы масштабируем объект перед его выводом на экран. Добавьте следующий код:

```
/// <summary>
/// Render the car given the current properties
/// </summary>
/// <param name="device">The device used to render the car</param>
public void Draw(Device device)
{
   // The car is a little bit too big, scale it down
   device.Transform.World = Matrix.Scaling(Scale,
      Scale, Scale) * Matrix.Translation(carLocation, Height, Depth);
   for (int i = 0; i < carMaterials.Length; i++)
   {
      device.Material = carMaterials[i];
      device.SetTexture(0, carTextures[i]);
      carMesh.DrawSubset(i);
   }
}
```

Перед тем как использовать класс car, необходимо создать локальные переменные, к которым может понадобиться общий доступ. Добавьте этот список параметров к вашему классу car:

```
// Public properties for car data
public float Location

   get ( return carLocation; }
   set ( carLocation = value; }

public float Diameter

   get ( return carDiameter; )

public float Speed

   get ( return carSpeed; (
   set ( carSpeed = value; 1

public bool IsMovingLeft

   get ( return movingLeft; }
   set ( movingLeft = value; I

public bool IsMovingRight

   get { return movingRight; 1
   set ( movingRight = value; }
```

Далее необходимо добавить переменную для поддержки класса car в основном движке игры. Добавьте следующие строки где-нибудь в разделе определения переменных приложения DodgerGame:

```
// Car
private Car car = null;
```

Поскольку мы используем устройство в качестве параметра конструктора, мы не можем создавать класс car до момента создания этого устройства. Было бы неплохо поместить алгоритм создания автомобиля в методе OnDeviceReset. Для этого сразу после создания Mesh-объекта road добавьте следующие строки, чтобы создать объект car в качестве устройства:

```
// Create our car
car = new Car (device);
```

После создания объекта класса car мы можем изменить алгоритм рендеринга, чтобы запустить процедуру отображения автомобиля. Сразу после вызова двух методов DrawRoad из метода OnPaint добавьте вызов рисунка из объекта car:

```
// Draw the current location of the car
car.Draw(device);
```

Итак, мы отобразили автомобиль на перемещающейся дороге. Дальше необходимо управлять движением автомобиля от одной стороны дороги к другой. Предположим, что пользователь имеет пока только клавиатуру и для управления автомобилем будет использовать ее. Для этих целей наиболее подходят клавиши с изображением стрелок. Перепишите метод OnKeyDown в классе DodgerGame, как указано в листинге 6.8.-

Листинг 6.8. Обработка команд с помощью клавиш.

```
/// <summary>
/// Handle key strokes
/// </summary>
protected override void OnKeyDown(System.Windows.Forms.KeyEventArgs e)
{
   // Handle the escape key for quiting
   if (e.KeyCode == Keys.Escape)
   {
      // Close the form and return
      this.Closed;

      return;
   }
   // Handle left and right keys
   if ((e.KeyCode == Keys.Left) || (e.KeyCode == Keys.NumPad4))
   {
      car.IsMovingLeft = true;
      car.IsMovingRight = false;
   }
   if ((e.KeyCode == Keys.Right) || (e.KeyCode == Keys.NumPad6))
   {
      car.IsMovingLeft = false;
      car.IsMovingRight = true;
   }
   // Stop moving
   if (e.KeyCode == Keys.NumPad5)
   {
      car.IsMovingLeft = false;
      car.IsMovingRight = false;
   }
}
```

Ничего особенного здесь не происходит. При нажатии на клавишу «Escape» игра будет закончена закрытием формы. Нажатием левой или правой клавиши-стрелки мы записываем соответствующее значение «true» в переменной перемещения «влево», «вправо» (в распечатке IsMovingLeft и IsMovingRight), при этом для перемещения в противоположную сторону данная переменная устанавливается в значение «false». Перемещение автомобиля прекращается, если нажата клавиша «5» на цифровой клавиатуре. Итак, при выполнении приложения нажатие этих клавиш заставит переменные изменять свои значения, но при этом сам автомобиль не будет двигаться. Необходимо также добавить функцию обновления для автомобиля. Добавьте метод, приведенный в листинге 6.9, в ваш класс car.

Листинг 6.9. Управление перемещением автомобиля.

```
/// <summary>
/// Update the cars state based on the elapsed time
/// </summary>
/// <param name="elapsedTime">Amount of time that has elapsed</param>
public void Update(float elapsedTime)
{
   if (movingLeft)
   {
      // Move the car
      carLocation += (carSpeed * elapsedTime);
      // Is the car all the way to the left?
      if (carLocation >= DodgerGame.RoadLocationLeft)
      {
         movingLeft = false;
         carLocation = DodgerGame.RoadLocationLeft;
      }
   }
   if (movingRight)
   {
      // Move the car
      carLocation -= (carSpeed * elapsedTime);
      // Is the car all the way to the right?
      if (carLocation <= DodgerGame.RoadLocationRight)
      {
         movingRight = false;
         carLocation = DodgerGame.RoadLocationRight;
      }
   }
}
```

Этот метод принимает в качестве параметра общее затраченное время «elapsed time» таким образом, чтобы мы могли поддерживать те же са-

мые перемещения на всех компьютерах. Сама функция достаточно проста. Если одна из переменных перемещения принимает значение «true», мы будем двигаться в соответствующем направлении (опираясь на параметр «elapsed time»). Затем проверяется местоположение автомобиля, если положение неправильное (автомобиль выходит за пределы дороги), движение останавливается полностью. Однако, в настоящий момент этот метод не вызывается, и для того чтобы его вызвать, необходимо изменить метод OnFrameUpdate в классе DodgerGame. Добавьте следующую строку к концу этого метода:

```
// Now that the road has been 'moved', update our car if it's moving
car.Update(elapsedTime);
```

Добавление препятствий

Поздравляем! Это — первое интерактивное графическое 3D-приложение, которое вы создали. Мы получили модель перемещения автомобиля или автомобильный симулятор. Несмотря на то, что на самом деле перемещается дорога (вниз относительно автомобиля), создается полное ощущение движения автомобиля. Таким образом, мы написали практически половину нашей игры. Теперь необходимо создать препятствия, которые вы потом будете объезжать. Подобно тому, как мы добавляли класс car, необходимо добавить новый класс «Obstacle» («Препятствия»). Проверьте, что вы включили директиву «using» Управляемого DirectX в этом новом файле кода.

Хотя предполагается использование одного Mesh-объекта для всех препятствий, было бы интересно внести некоторое разнообразие в список препятствий, например, различную форму, тип или цвет препятствий. Можно использовать или готовую заготовку Mesh-объекта, изменяя его тип, или материалы для изменения цвета. Таким образом, следует добавить константы и переменные, необходимые для класса obstacle, см. листинг 6.10.

Листинг 6.10. Константы класса «Obstacle».

```
// Object constants
private const int NumberMeshTypes = 5;
private const float ObjectLength = 3.0f;
private const float ObjectRadius = ObjectLength / 2.0f;
private const int ObjectStacksSlices = 18;
// obstacle colors
private static readonly Color[] ObstacleColors = (
   Color.Red, Color.Blue, Color.Green,
   Color.Bisque, Color.Cyan, Color.DarkKhaki,
```

```
    Color.OldLace, Color.PowderBlue, Color.DarkTurquoise,
    Color.Azure, Color.Violet, Color.Tomato,
    Color.Yellow, Color.Purple, Color.AliceBlue,
    Color.Honeydew, Color.Crimson, Color.Firebrick };
// Mesh information
private Mesh obstacleMesh = null;
private Material obstacleMaterial;
private Vector3 position;
private bool isleapot;
```

Как видно из первой константы, имеются пять различных типов Mesh-объектов (сфера, куб, тор, цилиндр, и заварочный чайник, соответственно, sphere, cube, torus, cylinder и teapot). Большинство этих типов будут иметь или параметр длины, или параметр радиуса, которыми можно варьировать для изменения размера препятствия. Существуют также дополнительные параметры Mesh-объекта (пачки, наборы, сектора, кольца и т.д), которые управляют числом полигонов (треугольники можно отнести к простейшим полигонам), составляющих объект. За это отвечает последняя константа в списке — *ObjectStacksSlices*. При увеличении значения этой константы растет число используемых полигонов, а следовательно, и качество картинки.

Далее идет список цветов для Mesh-объекта. Мы беспорядочно выбрали несколько цветов и включили их в список. Обратите внимание, что мы не сохраняем в этом классе ни массив материалов, ни текстуры для наших объектов. Как известно, заданный по умолчанию тип Mesh содержит только одно подмножество без текстур или материалов, поэтому эта дополнительная информация не нужна.

Поскольку препятствия «лежат» на движущейся дороге, необходимо согласовать перемещение препятствия с мировой системой координат. Необходимо связывать положение препятствия, которое будет изменяться с каждым кадром, с дорогой. Поэтому, поскольку мы не можем контролировать размер объекта «teapot», выбранного в качестве Mesh-объекта, в процессе его создания необходимо применить метод масштабирования.

Замените конструктор класса obstacle на следующий:

```
public Obstacle(Device device, float x, float y, float z)
{
    // Store our position
    position = new Vector3(x, y, z);
    // It's not a teapot
    isTeapot = false;
    // Create a new obstacle
    switch (Utility.Rnd.Next(NumberMeshTypes))
    {
```

```
    case 0:
        obstacleMesh = Mesh.Sphere(device, ObjectRadius, ObjectStacksSlices,
            ObjectStacksSlices);
        break;
case 1:
        obstacleMesh = Mesh.Box(device, ObjectLength, ObjectLength,
            ObjectLength);
        break;
case 2:
        obstacleMesh = Mesh.Teapot(device) ;
        isTeapot = true;
        break;
case 3:
        obstacleMesh = Mesh.Cylinder(device, ObjectRadius, ObjectRadius,
            ObjectLength,ObjectStacksSlices, ObjectStacksSlices);
        break;
    case 4:
        obstacleMesh = Mesh.Torus(device, ObjectRadius / 3.0f, ObjectRadius / 2.0f,
            ObjectStacksSlices, ObjectStacksSlices) ;
        break;
}
    // Set the obstacle color
    obstacleMaterial = new Material();
    Color objColor = ObstacleColors[Utility.Rnd.Next(ObstacleColors.Length)];
    obstacleMaterial.Ambient = objColor;
    obstacleMaterial.Diffuse = objColor;
}
```

Обратите внимание на использование здесь функции Rnd из модуля утилит. Исходник этой функции включен в CD диск. Ее задача заключается в том, чтобы просто возвращать случайные числа. Конструктор для нашего препятствия сохраняет заданное по умолчанию местоположение препятствия и значения по умолчанию для объекта Mesh. Затем он случайным образом выбирает один из типов Mesh-объекта и создает его. Наконец, он выбирает случайный цвет из списка и использует его как цвет материала для препятствий.

Прежде чем включить препятствие в движок игры, необходимо учесть пару моментов. Сначала необходимо создать функцию, которая изменяет положение объекта в соответствии с положением дороги, для этого добавим следующий код:

```
public void Update(float elapsedTime, float speed)
 {
position.Z += (speed * elapsedTime);
}
```

Снова используется параметр «elapsed time», чтобы обеспечить и согласовать различные скорости перемещений. Также пересылается текущее значение скорости дороги, чтобы совместить перемещение объекта и дороги. Для отображения перемещения препятствия необходимо использовать процедуру рендеринга. Добавьте метод, приведенный в листинге 6.11, к классу obstacle.

Листинг 6.11. Рисование препятствий.

```
public void Draw(Device device)
(
   if (isTeapot)
   {
      device.Transform.World = Matrix.Scaling(ObjectRadius, ObjectRadius,
      ObjectRadius)
         * Matrix.Translation(position);
   }
   else
   {
      device.Transform.World = Matrix.Translation(position);
   }
   device.Material = obstacleMaterial;
   device.SetTexturef0, null);
   obstacleMesh.DrawSubset(0);
}
```

Поскольку Mesh-объект «teapot» после создания не масштабируется (если вы рисуете один из этих объектов), необходимо сначала смасштабировать этот объект и уже затем переместить его в соответствующую позицию. Затем устанавливается материал для цвета объекта, присваивается значение «null» для текстуры, и рисуется Mesh-объект.

- Очевидно, потребуется иметь не одно препятствие на дороге. Поэтому нам понадобится простой метод добавления и удаления препятствий. Использование массива возможно, но нецелесообразно, поскольку неоднократное изменение размера массива несколько громоздко. Лучше создать класс-коллекцию для хранения данных о препятствиях. Добавьте класс, приведенный в листинге 6.12, в конец вашего файла с кодом препятствия.

Листинг 6.12. Класс-коллекция Препятствий, класс obstacles.

```
public class Obstacles : IEnumerable
{
   private ArrayList obstacleList = new ArrayList();
```

```
    /// <summary>
    /// Indexer for this class
    /// </summary>
    public Obstacle this[int index]
    {
        get
        {
            return (Obstacle)obstacleList[index] ;
        }

    // Get the enumerator from our arraylist
    public IEnumerator GetEnumerator0
    {
        return obstacleList.GetEnumerator();
    }
    /// <summary>
    /// Add an obstacle to our list
    /// </summary>
    /// <param name="obstacle">The obstacle to add</param>
    public void Add(Obstacle obstacle)

    {
        obstacleList.Add(obstacle);
    }
    /// <summary>
    /// Remove an obstacle from our list
    /// </summary>
    /// <param name="obstacle">The obstacle to remove</param>
    public void Remove(Obstacle obstacle)
    {
        obstacleList.Remove(obstacle);
    }
    /// <summary>
    /// Clear the obstacle list
    /// </summary>
    public void Clear()
    {
        obstacleList.Clear0 ;
    }
}
```

Для правильной компиляции необходимо поместить директиву «using» для System.Collections в самом начале файла кода для этого класса. Этот класс имеет индексированный прямой доступ к объекту препятствия, а также метод перебора и следующие три действия: добавление, удаление и очистка, соответственно, add, remove и clear. Имея эти базовые возможности, можно приступить к добавлению объекта препятствия.

Сначала необходимо внести переменную, которую можно использовать для поддержки списка текущих препятствий в сцене. Добавьте следующую переменную в класс DodgerGame:

```
// Obstacle information
private Obstacles obstacles;
```

Далее необходимо создать функцию заполнения следующей дорожной секции новыми препятствиями. Это будет выглядеть следующим образом:

```
/// <summary>
/// Add a series of obstacles onto a road section
/// </summary>
/// <param name="minDepth">Minimum depth of the obstacles</param>
private void AddObstacles(float minDepth)
{
   // Add the right number of obstacles
   int numberToAdd = (int)((RoadSize / car.Diameter - 1) / 2.0f);
   // Get the minimum space between obstacles in this section
   float minSize = ((RoadSize / numberToAdd) - car.Diameter) / 2.0f;
   for (int i = 0; i < numberToAdd; i++)
   {
      // Get a random # in the min size range
      float depth = minDepth - ((float)Utility.Rnd.NextDouble() * minSize);
      // Make sure it's in the right range
      depth -= (i * (car.Diameter * 2));
      // Pick the left or right side of the road
      float location = (Utility.Rnd.Next(50) >
      25)?RoadLocationLeft:RoadLocationRight;
      // Add this obstacle
      obstacles.Add(new Obstacle(device, location, ObstacleHeight, depth));
   }
}
```

Эта функция является отправной точкой для создания препятствий в игре. Вначале она вычисляет число препятствий, которые необходимо добавить в данной дорожной секции. Необходимо убедиться в том, что между препятствиями имеется достаточно пространства для автомобиля. После того как функция определит число препятствий и минимальное пространство между ними, она случайным образом помещает их в участок дороги, после чего добавляет их к текущему списку препятствий. Обратите внимание на константу *ObstacleHeight,* высоту препятствия, используемую при создании нового препятствия:

```
private const float ObstacleHeight = Car.Height * 0.S5f;
```

→ *ПК* 604

Осталось сделать три вещи перед тем, как препятствия появятся в сцене: необходимо добавить обращение к методу добавления препятствий, необходимо убедиться, что функция обновления вызывается для каждого препятствия в сцене, и необходимо отобразить препятствия на экране. Кроме того, для сброса переменных при запуске новой игры необходимо создать соответствующую функцию и использовать данный алгоритм при первом вызове метода AddObstacles. Добавьте метод, приведенный в листинге 6.13.

Листинг 6.13. Загрузка игровых опций, заданных по умолчанию.

```
/// <summary>
/// Here we will load all the default game options
/// </summary>
private void LoadDefaultGameOptions()
{
   // Road information
   RoadDepth0 = 0.0f;
   RoadDepth1 = -100.0f;
   RoadSpeed = 30.0f;
   // Car data information
   car.Location = RoadLocationLeft;
   car.Speed = 10.0f;
   car.IsMovingLeft = false;
   car.IsMovingRight = false;
   // Remove any obstacles currently in the game
   foreach(Obstacle o in obstacles)
   {
      // Dispose it first
      o.Disposed;
   }
   obstacles.Clear d;
   // Add some obstacles
   AddObstacles(RoadDepth1);
   // Start our timer
   Utility.Timer(DirectXTimer.Start);
}
```

Этот метод принимает различные значения переменных, которые можно при необходимости установить по умолчанию. Он также принимает любые существующие препятствия, находящиеся в списке, располагает их и очищает список перед заполнением новыми препятствиями. И наконец, он запускает таймер, после чего необходимо добавить вызов созданной функции после создания устройства в методе InitializeGraphics. Не

добавляйте (!) эту функцию в метод OnDeviceReset; мы хотим вызывать эту функцию, только когда начинается новая игра.

```
// Load the default game options
LoadDefaultGameOptions();
```

Затем нужно добавить вызов в методе OnFrameUpdate для обновления препятствий при каждой смене кадра. Таким образом, перед методом обновления объекта car добавьте следующий код в метод OnFrameUpdate:

```
// Move our obstacles
foreach(Obstacle o in obstacles)
{
   // Update the obstacle, check to see if it hits the car
   o.Update(elapsedTime, RoadSpeed);
}
```

Последнее, что необходимо выполнить, это отобразить препятствия на экране. Для этого в методе OnPaint в конце метода прорисовки автомобиля необходимо добавить следующий код:

```
// Draw any obstacles currently visible
foreach(Obstacle o in obstacles)
{
   o.Draw(device);
}
```

Теперь попытаемся запустить игру! Вы проехали часть дороги (естественно, перемещая саму дорогу), проехали несколько препятствий, что дальше? После этих нескольких препятствий больше ничего не появляется. Вспомните, метод AddObstacles добавляет препятствия только к одной секции дороги! Таким образом, перемещая дорожные секции, мы не вызываем повторно метод создания препятствий для этих «новых» дорожных секций. Перепишите секцию кода, который используется для добавления новых дорожных секций, следующим образом:

```
// Check to see if we need to cycle the road
if (RoadDepth0 > 75.0f)
{
   RoadDepth0 = RoadDepth1 - 100.0f;
   AddObstacles(RoadDepth0);
}
   if (RoadDepth1 > 75.0f)
   {
```

```
    RoadDepthl = RoadDepth0 - 100.0f;
    AddObstacles(RoadDepthl);
}
```

Теперь это выглядят более реалистично. У нас есть автомобиль, перемещающийся мимо препятствий (пока проносясь через препятствия). К самим препятствиям также можно добавить движение, например, заставить их вращаться. Для этого вначале необходимо добавить несколько новых переменных к классу obstacle, чтобы управлять вращением препятствий:

```
// Rotation information
private float rotation = 0;
private float rotationspeed = 0.0f;
private Vector3 rotationVector;
```

Скорость, с которой они вращаются, и оси вращения должны выбираться случайным образом. Это легко осуществить, добавляя следующие строки в конец конструктора класса obstacle:

```
rotationspeed = (float)Utility.Rnd.NextDouble() * (float)Math.PI;
rotationVector = new Vector3( (float)Utility.Rnd.NextDouble() ,
    (float) Utility.Rnd.NextDouble(), (float) Utility.Rnd.NextDouble ()) ;
```

Осталось два момента, необходимых для того, чтобы задать правильное вращение препятствиям. Для начала необходимо включить вращение в функцию обновления (update function) следующим образом:

```
rotation += (rotationspeed * elapsedTime);
```

Ничего необычного, просто скорость вращения увеличивается с учетом значения elapsedTime и текущей скорости вращения, выбранной случайным образом. Добавляем вращение в наше преобразование мировых координат, преобразовывая координаты следующим образом:

```
if (isTeapot)
{
   device.Transform.World = Matrix.RotationAxis(rotationVector, rotation)
      * Matrix.Scaling(ObjectRadius, ObjectRadius, ObjectRadius)
      * Matrix.Translation(position);
}
else
{
   device.Transform.World = Matrix.RotationAxis(rotationVector, rotation)
         * Matrix.Translation(position);
}
```

Теперь, при запуске игры, видны препятствия, беспорядочно вращающиеся по мере увеличения или уменьшения скорости автомобиля. Каков следующий шаг? Необходимо добавить опции, позволяющие отслеживать состояние игры и текущий счет, который необходимо обнулять в начале новой игры и увеличивать после удачного прохождения препятствия. Добавьте следующие значения переменных объекта к движку игры «main» в классе DodgerGame:

```
// Game information
private bool isGameOver = true;
private int gameOverlick = 0;
private booi hasGameStarted = false;
private int score = 0;
```

Здесь сохраняется вся информация об игре. Вам необходимо знать, началась или закончилась игра, время окончания последней игры и текущий счет. Было бы желательно учесть также скорость перемещения препятствий и увеличивать текущий счет в зависимости от сложности игры. Первое, что необходимо для этого сделать, — добавить следующие строки для сброса счета очков в опции LoadDefaultGameOptions, чтобы в начале новой игры все состояния обнулялись:

```
car.IsMovingRight = false;
score = 0;
```

В методе OnFrameUpdate до того момента, как вы начинаете перемещать препятствия, добавьте следующий код:

```
// Remove any obstacles that are past the car
// Increase the score for each one, and also increase
// the road speed to make the game harder.
•Obstacles removeObstacles = new Obstacles));
foreach(Obstacle o in obstacles)
{
   if (o.Depth > car.Diameter - (Car.Depth * 2))
   (
      // Add this obstacle to our list to remove
      removeObstacles.Add(o);
      // Increase roadspeed
      RoadSpeed += RoadSpeedIncrement;
      // Make sure the road speed stays below max
      if (RoadSpeed >= MaximumRoadSpeed)
      {
         RoadSpeed = MaximumRoadSpeed;
      }
      // Increase the car speed as well
```

```
            car.IncrementSpeed();
            // Add the new score
            score += (int)(RoadSpeed * (RoadSpeed / car.Speed));
        }
    }
    // Remove the obstacles in the list
    foreach(Obstacle o in removeObstacles)
    {
        obstacles.Remove(o);
        // May as well dispose it as well
        o.Disposed ;
    }
    removeObstacles.Clear();
```

Это необходимо, чтобы получить список препятствий, которые вы уже прошли. Для каждого препятствия необходимо увеличить значение счета с учетом текущей скорости перемещения дороги или автомобиля. После того как игра завершилась, препятствия удаляются из списка. Как вы можете видеть, при вычислении счета используется формула, учитывающая текущую дорожную скорость, таким образом, пройдя большее расстояние, вы получите большее количество очков. Обратите также внимание на метод, использующийся для увеличения скорости автомобиля:

```
/// <summary>
/// Increment the movement speed of the car
/// </summary>
public void IncrementSpeed()
{
    carSpeed += SpeedIncrement;
}
```

Теперь следует добавить в класс obstacle новый метод для определения момента, когда автомобиль врезается в одно из препятствий:

```
public bool IsHittingCar(float carLocation, float carDiameter)
{
    // In order for the obstacle to be hitting the car,
    // it must be on the same side of the road and
    // hitting the car
    if (position.Z > (Car.Depth - (carDiameter / 2.0f)))
    {
        // are we on the right side of the car
        if ((carLocation < 0) && (position.X < 0))
            return true;
        if ((carLocation > 0) && (position.X > 0))
            return true;
```

```
    }
    return false;
}
```

Довольно просто. Вы выясняете, если автомобиль находится в той же самой глубине секции и с той же самой стороны дороги, что и препятствие, это означает столкновение с препятствием, и возвращается значение «true». В противном случае возвращается значение «false», автомобиль успешно преодолевает препятствие. Теперь необходимо вставить этот код в движок игры. Замените код обновления препятствия более совершенным кодом:

```
// Move our obstacles
foreach(Obstacle o in obstacles)
{
    // Update the obstacle, check to see if it hits the car
    o.Update(elapsedTime, RoadSpeed);
    if (o.IsHittingCar(car.Location, car.Diameter))
    {
        // If it does hit the car, the game is over.
        isGameOver = true;
        gameOverTick = System.Environment.TickCount;
        // Stop our timer
        Utility.Timer(DirectXTimer.Stop);
    }
}
```

Теперь, после того как вы «обнаружили» столкновение с препятствием, игра заканчивается. Вы фиксируете состояние игры и останавливаете таймер.

Последние штрихи

На данный момент мы ничего не делаем с переменными состояния. В первую очередь осуществляем логику окончания игры. Очередной запуск игры необходимо привязать к нажатию любой клавиши. Для этого после окончания игры (случай, когда мы потерпели неудачу) выполняется небольшая пауза (около одной секунды), и затем нажатие любой клавиши запускает игру еще раз. Необходимо также проверить, что в момент окончания игры не обновляются никакие другие состояния. Таким образом, самые первые строки в методе OnFrameUpdate должны быть следующими:

```
// Nothing to update if the game is over
if ((isGameOver) || (!hasGameStarted))
    return;
```

Теперь необходимо обработать нажатия клавиш, чтобы перезапустить игру. В конец перегрузки OnKeyDown Вы можете добавить следующую логику:

```
if (isGameOver)
{
   LoadDefaultGameOptions();
}
// Always set isGameOver to false when a key is pressed
isGameOver = false;
hasGameStarted = true;
```

Это и есть поведение, которого мы добивались. Теперь после окончания игры игрок нажимает клавишу, и новая игра начинается с заданных по умолчанию игровых опций. При желании вы можете удалить вызов опции LoadDefaultGameOptions из метода InitializeGraphics, поскольку она будет вызываться автоматически после нажатия клавиши. Однако, у нас пока нет кода, который вызовет небольшую паузу после прекращения игры. Вы можете поместить его в процедуру OnKeyDown или сразу после проверки нажатия клавиши «Escape»:

```
// Ignore keystrokes for a second after the game is over
if ((System.Environment.TickCount - gameOverTick) < 1000)
{
   return;
}
```

Выполнение данных строк игнорирует любые нажатия клавиши (кроме клавиши выхода Escape) в течение одной секунды после того, как игра закончена. Теперь вы можете приступить к игре. Хотя можно еще добавить дополнительный текст, отображающий состояние игры или некоторые комментарии. В составе имен («namespace») приложения Direct3D имеется класс шрифтов («Font class»), который может использоваться для отображения текста, однако, Font class имеется и в пространстве имен System.Drawing. Эти два класса будут конфликтовать, если вы попытаетесь использовать «Font» без идентификации. К счастью, вы можете переименовать данный объект, используя директиву «using» следующим образом:

```
using Direct3D = Microsoft.DirectX.Direct3D;
```

Создаваемые шрифты могут быть различного цвета, но должны иметь единый размер и семейство. При желании можно отображать два различных типа текста или шрифта для этой игры, но это значит, что их необходимо обозначить, добавив следующие переменные к классу DodgerGame:

```
// Fonts
private Direct3D.Font scoreFont = null;
private Direct3D.Font gameFont = null;
```

Необходимо инициализировать эти переменные, и важно, чтобы это было сделано после создания устройства. Нет необходимости делать это в методе OnDeviceReset, поскольку эти объекты автоматически обработают сброс устройства. Добавьте следующие строки в конце метода InitializeGraphics:

```
// Create our fonts
scoreFont = new Direct3D.Font (device, new System.Drawing.Font("Arial",
   12.Of,
   FontStyle.Bold));
gameFont = new Direct3D.Font(device, new System.Drawing.Font("Arial",
   36.Of,
   FontStyle.Bold ! FontStyle.Italic));
```

Теперь у нас имеются два шрифта различных размеров, но одного и того же типа Arial. Далее необходимо переписать код рендеринга для отображения текста. Если вы хотите отобразить текст в конце игры, добавьте следующий код в конце метода прорисовки автомобиля:

```
if (hasGameStarted)
{
   // Draw our score
   scoreFont.DrawText (null, string. Format ("Current score: (Of, score),
      new Rectangle(5,5,0,0), DrawTextFormat.NoClip, Color.Yellow) ;
}
if (isGameOver)
{
   // If the game is over, notify the player
   if (hasGameStarted)
   {
      gameFont.DrawText (null, "You crashed. The game is over.", new
         Rectangle(25,45,0,0),
         DrawTextFormat.NoClip, Color.Red);
   }
   if ((System.Environment.TickCount - gameOverTick) >= 1000)
   {
      // Only draw this if the game has been over more than one second
      gameFont.DrawText(null, "Press any key to begin.", new
      Rectangle(25,100,0,0),
         DrawTextFormat.NoClip, Color.WhiteSmoke);
   }
}
```

Алгоритм отображения текста DrawText будет обсуждаться подробнее в следующей главе. В отображаемых текстах можно показать любую информацию, например: текущий счет, сообщение о неудаче, сообщение о начале игры после нажатия любой клавиши и т.д.

Итак, теперь у нас имеется законченный вариант игры. Вы можете запустить ее, фиксировать изменение счета, конец игры, неудачу, и затем начать игру сначала. Что осталось? Хорошо было бы сохранять лучший результат, который бы постоянно пересохранялся при более высоком результате.

ДОБАВЛЕНИЕ КОММЕНТАРИЯ «High Scores» — наилучший результат

Нас в первую очередь будут интересовать имена нескольких игроков и их максимальный результат. Создадим для этого простую структуру, добавив следующий код в главное пространство имен игры:

```
/// <summary>
/// Structure used to maintain high scores
/// </summary>
public struct HighScore
{
    private int realScore;
    private string playerName;
    public int Score { get { return realScore; } set { realScore = value; } }
    public string Name { get { return playerName; } set { playerName = value; } }
}
```

Дальше необходимо установить список результатов «high scores» в нашем движке игры. Мы рассмотрим формирование списка только для трех игроков, используя при этом массив для хранения этих данных. Добавьте следующие строки в движок игры:

```
// High score information
private HighScore[] highScores = new HighScore[3];
private string defaultHighScoreName = string.Empty;
```

Теперь создадим три отдельные функции. Первая проверит текущий счет и сравнит его с имеющимся, чтобы внести при необходимости новое значение в список. Следующая функция сохранит информацию о наилучшем результате в системный реестр, а последняя будет вызывать это значение из системного реестра. Добавьте соответствующую программу в движок игры:

```
/// <summary>
/// Check to see what the best high score is. If this beats it,
/// store the index, and ask for a name
/// </summary>
private void CheckHighScore()
{
   int index = -1;
   for (int i = highScores.Length - 1; i >= 0; i--)
   {
      if (score >= highScores[i].Score) // We beat this score
      {
         index = i;
      }
   }
   // We beat the score if index is greater than 0
   if (index >= 0)
   {
      for (int i = highScores.Length - 1; i > index ; i--)
      {
         // Move each existing score down one
         highScores[i] = highScores[i-1];
      }
      highScores[index].Score = score;
      highScores[index].Name = Input.InputBox("You got a high score!!",
         "Please enter your name.", defaultHighScoreName);
   }
}
/// <summary>
/// Load the high scores from the registry
/// </summary>
private void LoadHighScores()
{
   Microsoft.Win32.RegistryKey key =
      Microsoft.Win32.Registry.LocalMachine.CreateSubKey(
      "Software\\MDXBoox\\Dodger");
   try
   {
      for(int i = 0; i < highScores.Length; i++)
      {
         highScores[i].Name = (string)key.GetValue(
            string.Format("Player{0}", i), string.Empty);
         highScores[i].Score = (int)key.GetValue(
            string.Format("Score(0)", i), 0);
      }
      defaultHighScoreName = (string)key.GetValue(
         "PlayerName", System.Environment.UserName);
   }
```

```
    finally
    {
        if (key != null)
        (
            key.Closed; // Make sure to close the key
        }
    }
}
/// <summary>
/// Save all the high score information to the registry
/// </summary>
public void SaveHighScores ()
{
    Microsoft.Win32.RegistryKey key =
        Microsoft.Win32.Registry.LocaiMachine.CreateSubKey(
        "Software\\MDXBoox\\Dodger");
    try
    {
        for(int i = 0; i < highScores.Length; it++)
        {
            key.SetValue(string.Format("Player{0}", i), highScores[i].Name);
            key.SetValue(string.Format("Score{O}", i), highScores[i].Score);
        }
        key.SetValue("PlayerName", defaultHighScoreName);
    }
    finally
    {
        if (key != null)
        {
            key.Close(); // Make sure to close the key
        }
    }
}
```

Не будем слишком глубоко вникать в конструкцию этих функций, поскольку они имеют дело главным образом со встроенными классами .NET и не воздействуют на код Управляемого DirectX. Однако, важно показать, откуда эти методы вызываются в движке игры.

Проверка значения для «high scores» должна производиться сразу после окончания игры. Замените код в методе OnFrameUpdate, который проверяет факт столкновения автомобиля с препятствием, на следующий:

```
if (o.IsHittingCar(car.Location, car.Diameter))
{
```

```
    // If it does hit the car, the game is over.
    isGameOver = true;
    gameOverTick = System.Environment.TickCount;
    // Stop our timer
    Utility.Timer(DirectXTimer.Stop);
    // Check to see if we want to add this to our high scores list
    CheckHighScore();
}
```

Вы можете загружать значение «high scores» в конце конструктора основного движка игры. Следует отметить, что алгоритм сохранения является общим (тогда как другие методы являются индивидуальными). Поэтому мы будем вызывать его из нашего основного кода. Перепишите основной метод «main»:

```
using (DodgerGame frm = new DodgerGame())
{
    // Show our form and initialize our graphics engine
    frm.Show();
    frm.InitializeGraphics();
    Application.Run(frm);
    // Make sure to save the high scores
    frm.SaveHighScoresO ;
}
```

И заключительный момент действия — отображение на экране список наилучших результатов. Мы добавим его в наш код рендеринга. До того как вызвать окончательный метод для нашего текста, добавим к секции кода рендеринга надписи «High scores» следующие строки:

```
// Draw the high scores
gameFont.DrawText(null, "High Scores: ", new Rectangle(25,155,0,0),
    DrawTextFormat.NoClip, Color.CornflowerBlue);
for (int i = 0; i < highScores.Length; i++)
{
    gameFont.DrawText(null, string.Format("Player; {0} : {1}",
        highScores[i].Name,
        highScores[i].Score),
        new Rectangle(25,210 + (i * 55),0,0), DrawTextFormat.NoClip,
            Color.CornflowerBlue);
            }
```

Поздравляем, вы только что закончили написание вашей первой игры.

Краткие выводы

В этой главе мы проделали следующее.

Использовали объекты Mesh для рендеринга игровых объектов.

- Проверили некоторые возможности устройства.
- Освоили простейший пользовательский ввод.
- Создали систему подсчета очков.
- Объединили все в общую конструкцию.

Результаты игры приведены на **рис.** 6.1.

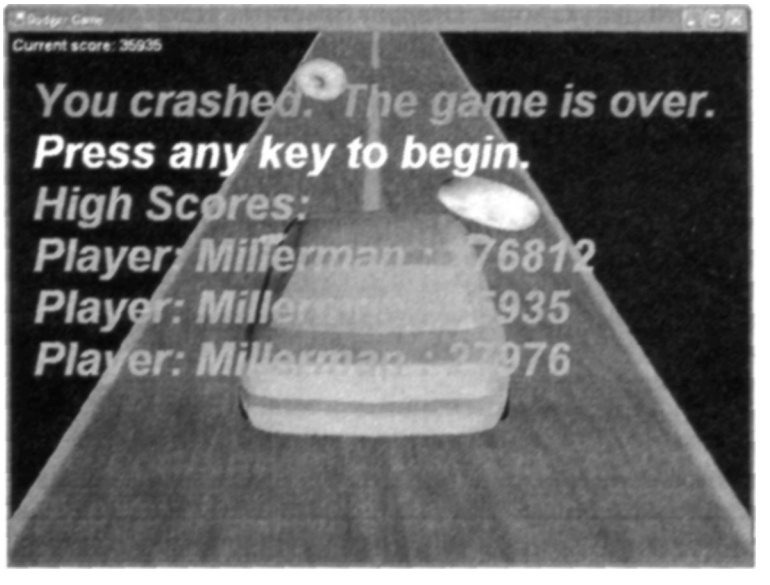


Рис.6.1. Завершенная игра

В следующей главе мы обсудим использование более совершенных возможностей и свойств mesh-объектов.

ЧАСТЬ II

ОСНОВНЫЕ КОНЦЕПЦИИ ПОСТРОЕНИЯ ГРАФИКИ

Глава 7. Использование дополнительных свойств и возможностей Mesh-объектов

Глава 8. Введение в ресурсы

Глава 9. Применение других типов Mesh

Глава 10. Использование вспомогательных классов

Глава 7. Использование дополнительных свойств и возможностей Mesh-объектов

В данной главе мы обсудим более широкие возможности применения объектов «Mesh», включая.

- Оптимизацию данных Mesh-объекта.
- Упрощение Mesh-объектов.
- Создание Mesh с новыми компонентами данных о вершинах.
- Объединение вершин.

Создание копий Mesh-объектов

На данном этапе изучения книги мы имеем возможность загружать Mesh-объекты и отображать их на экране. Наша сцена имеет несколько источников света, и, тем не менее, объекты кажутся темными. Рассматривая свойства объектов, можно заметить, что вершинные форматы не включают данные нормалей к поверхностям, необходимые для вычисления более реалистичного освещения.

В данном случае необходимо взять все имеющиеся данные Mesh-объекта и добавить к ним данные нормалей. Механизм для осуществления данной операции приведен в листинге 7.1:

Листинг 7.1. Добавление нормалей к Mesh-объекту.

```
// Check if mesh doesn't include normal data
if ((mesh.VertexFormat & VertexFormats.Normal) != VertexFormats.Normal)
{
   Mesh tempMesh = mesh.Clone(mesh.Options.Value,
      mesh.VertexFormat | VertexFormats.Normal, device);
   tempMesh.ComputeNormals();
   // Replace existing mesh
   mesh.Disposed ;
   mesh = tempMesh;
}
```

Здесь мы берем существующий Mesh-объект и определяем, содержит ли он данные нормалей или нет. Значение вершинного формата VertexFormat возвращает список параметров VertexFormats, которые объединены через логический оператор «ИЛИ», поэтому мы используем оператор «И», чтобы определить, установлен ли бит нормали. В случае если он не установлен, мы создаем с помощью функции Clone второй, вре-

менный Mesh-объект, являющийся копией первоначального объекта. Данный метод имеет три варианта загрузки:

```
public Microsoft.DirectX.Direct3D.Mesh Clone (
   Microsoft.DirectX.Direct3D.MeshFlags options ,
   Microsoft.DirectX.Direct3D.GraphicsStream declaration ,
   Microsoft.DirectX.Direct3D.Device device )

public Microsoft.DirectX.Direct3D.Mesh Clone (
   Microsoft.DirectX.Direct3D.MeshFlags options ,
   Microsoft.DirectX.Direct3D.VertexEleraent[] declaration ,
   Microsoft.DirectX.Direct3D.Device device )

public Microsoft.DirectX.Direct3D.Mesh Clone (
   Microsoft.DirectX.Direct3D.MeshFlags options ,
   Microsoft.DirectX.Direct3D.VertexFormats vertexFormat ,
   Microsoft.DirectX.Direct3D.Device device )
```

В нашем примере мы использовали последнюю перегрузку. В каждой из перегрузок, первые и последние параметры — те же самые. Параметр опций — *options* позволяет вновь созданному Mesh-объекту иметь различный набор дополнительных опций. Вы можете сохранить те же самые опции в новом объекте-копии (то, что мы делали ранее), а можете изменить их. Было бы предпочтительнее, чтобы новый Mesh-объект постоянно находился в системной памяти, нежели в управляемой памяти. Любой из указателей MeshFlags, которые являются доступными при создании объекта, также доступны в течение операции создания копии.

Последний параметр метода Clone *device* — устройство, на базе которого будет создан новый объект. В большинстве случаев это будет то же самое устройство, которое вы создаете для первоначального объекта, но возможно также и создание копий объекта под абсолютно различные устройства. Например, скажем, вы написали приложение, которое было рассчитано на несколько мониторов, каждый из которых работал в полноэкранном режиме. Если Mesh-объект отображался на первом мониторе, не было необходимости иметь «экземпляр» этого объекта на втором. При желании отобразить объект на втором мониторе, вы можете легко продублировать его (или другими словами создать и отобразить копию объекта) на новое устройство и иметь к нему доступ. Наш предыдущий пример просто использовал уже имеющееся в наличии устройство.

Параметр *declaration* в процедуре Clone определяет, каким образом данные будут отображаться на экране. В используемом нами варианте мы пропускаем описание вершинного формата вновь созданной копии объекта (которая может отличаться от оригинала). В других перегрузках

данный параметр предназначен для определения вершин (Vertex declarations) созданного объекта.

Как вы можете видеть, это — вспомогательная функция класса Mesh, которая может автоматически вычислять нормали Mesh-объектов. В неиспользуемых нами вариантах загрузки присутствует параметр, который используется для ввода информации, имеющий вид массива целых чисел или потока GraphicsStream.

Вы можете использовать возможность создания копии объекта в целях создания новых экземпляров того же Mesh-объекта для нового устройства или изменять опции уже созданного объекта. Вы можете также использовать это для создания объекта с новыми значениями формата вершины, или даже удалять существующие данные формата вершины из вашего Mesh-объекта. Если объект, который есть на данный момент, не удовлетворяет вашим требованиям, вы можете имитировать его, создав копию, и изменить эту копию по необходимости.

Оптимизация данных Mesh-объекта

Создание копии Mesh-объекта — не единственный способ расширить свойства имеющегося объекта. Существует еще несколько способов оптимизации Mesh-обьектов. Функция Optimize для объекта похожа на метод создания копии, описанный выше (данная функция позволяет создавать новый Mesh-объект с различными опциями), однако, она может также выполнять оптимизацию в процессе создания нового объекта. Следует отметить, что мы не можем использовать функцию Optimize для добавления или удаления вершинных данных или для дублирования их под новое устройство. Рассмотрим основную перегрузку метода Optimize:

```
public Microsoft.DirectX.Direct3D.Mesh Optimize (
   Microsoft.DirectX.Direct3D.MeshFlags flags ,
   int[ ] adjacencyln , out int[ ] adjacencyOut ,
   out int[ ] faceRemap ,
   out Microsoft.DirectX.Direct3D.GraphicsStream vertexRemap )
```

Каждая из четырех перегрузок этого метода принимает одинаковый набор аргументов, или просто флажков и вводимых данных. Обратите внимание, что параметр *adjacencyln* может использоваться или как массив целых чисел (как показано выше), или как поток данных GraphicsStream.

Параметр флажков используется, чтобы определить, каким может быть создан новый объект. Данный параметр может иметь разное количество флажков из списка MeshFlags (за исключением флажков Use32Bit или WriteOnly). Флажки Optimize, которые могут использоваться специальным образом для этой функции, приведены в таблице 7.1:

Таблица 7.1. Функции оптимизации Mesh-объекта

MeshFlags.OptimizeCompact	Переупорядочивает поверхности в Mesh-объекте, чтобы удалить неиспользованные вершины и поверхности
MeshFlags.OptimizeAttrSort	Переупорядочивает поверхности в Mesh-объекте так, чтобы иметь меньшее количество изменений состояния аттрибута, который может улучшить выполнение DrawSubset
MeshFlags.OptimizeDeviceIndependent	Использование этой функции затрагивает размер кэша вершины, определяя заданный по умолчанию размер кэша, и обеспечивает достаточно эффективное выполнение на используемых аппаратных средствах
MeshFlags.OptimizeDoNotSplit	Использование этого флажка определяет, что вершины не должны быть разбиты в случае, если они распределены между группами атрибутов
MeshFlags.OptimizeIgnore Verts	Оптимизирует только поверхности, игнорируя вершины
MeshFlags.OptimizeStripeReorder	Использование этого флажка переупорядочивает поверхности, чтобы максимизировать длину смежных треугольников или полигонов
MeshFlags. Optimize VertexCache	Использование этого флажка переупорядочивает поверхности для увеличения скорости кэширования вершины

ОПТИМИЗАЦИЯ MESH-ОБЪЕКТОВ НА МЕСТЕ

Что если вы не хотите создавать новый Mesh-объект целиком? Вы не хотите изменять различные флажки в процессе создания, вы только хотите использовать преимущества, полученные при оптимизации объекта.

Существует похожий метод для Mesh-объекта, называемый OptimizeInPlace, который принимает те же самые параметры, только с двумя различиями. Флаговый параметр должен быть одним из

флажков оптимизации (вы не можете изменять ни один из флажков создания), и еще, этот метод не имеет возвращаемого значения. Все оптимизации будут проходить непосредственно для объекта, из которого вызывается данная процедура.

Также понадобится параметр AdjacencyIn, он может быть представлен в виде целочисленного массива, включающего в себя три целых числа на каждую сторону, которые определяют трех смежных «соседей» каждой поверхности в объекте, или же в виде потока графических данных, в котором содержится та же самая информация.

Если вы выбираете одну из двух последующих перегрузок, последние три параметра в ней представляют те данные, которые будут возвращены вам посредством параметра out. Первым значением этих возвращенных данных будет новая информация смежности. Второе значение — новый индекс для каждой поверхности в объекте. Последним значением будет графический поток, который содержит новый индекс для каждой вершины. Многие из приложений не потребуют этой информации и пропустят эти функции.

Следующая небольшая секция кода показывает сортировку объекта с использованием буфера атрибутов и обеспечивает его размещение в управляемой памяти:

```
// Compact our mesh
Mesh tempMesh = mesh.Optimize
    (MeshFlags.Managed |
        MeshFlags.OptimizeAttrSort |
        MeshFlags.OptimizeDoNotSplit, adj) ;
mesh.Dispose();
mesh = tempMesh;
```

ФОРМИРОВАНИЕ ИНФОРМАЦИИ СМЕЖНОСТИ

Обратите внимание, что многие из более совершенных методов Mesh используют информацию смежности. Вы можете получить эту информацию в процессе создания Mesh-объекта. Но если объект уже создан, то для получения этой информации существует функция GenerateAdjacency. Первый параметр этой функции — число с плавающей точкой, которое можно использовать для того, чтобы определить те вершины, местоположение которых отличается менее чем на это значение, как смежные или совпадающие вершины. Второй параметр— целочисленный массив, который будет заполнен информацией смежности. Этот массив должен иметь размер по крайней мере 3 * mesh.NumberFaces.

Упрощение существующих Mesh-объектов

Теперь предположим, что ваш художник только что нарисовал вам некоторый объект в вашей сцене, который находится в различных положениях в зависимости от того, на каком уровне вы находитесь. На некоторых уровнях объект располагается на заднем плане и не требует столь же детальной деталировки, как на других уровнях. Естественно, можно дать задание художнику, что бы он сделал вам две различные модели, с высоким и низким качеством деталировки, но есть и более простое решение — можно использовать некоторые встроенные упрощения для имеющегося класса Mesh.

Программа упрощения изменяет существующий объект. Суть метода заключается в том, чтобы, используя набор весовых параметров, попробовать удалить как можно больше поверхностей и вершин объекта для получения низко-детального объекта. Однако, прежде чем упростить объект, его необходимо очистить. Для этого добавляют другую вершину, которая является общей для двух треугольников, расположенных веером (мы уже рассматривали такой тип примитива). Рассмотрим процедуру Clean:

```
public static Microsoft.DirectX.Direct3D.Mesh Clean (
   Microsoft.DirectX.Direct3D.Mesh mesh ,
   Microsoft.DirectX.Direct3D.GraphicsStream adjacency ,
   Microsoft.DirectX.Direct3D.GraphicsStream adjacencyOut ,
   out System.String errorsAndWarnings )
```

Обратите внимание, это напоминает более ранние функции. Процедура аналогичным образом принимает параметр объекта, который мы собираемся очистить, и информацию смежности. Однако, помимо этого здесь требуется параметр смежности adjacencyOut. Наиболее общая методика должна использовать смежные данные в графическом потоке, который вы получаете при создании объекта, также как параметры adjacencyIn и adjacencyOut для очистки. Имеется также возвращаемая строка, которая позволяет выводить сообщения об ошибках или предупреждениях, с которыми можно столкнуться при очистке объекта. Следует также обратить внимание на то, что параметры смежности могут быть или в виде графических потоков, как показано выше, или в виде массивов целых чисел.

Чтобы показать последствия выполнения программы упрощения, использующей эти методы, можно взять пример, который приводился в юнце главы 5, и попробовать использовать его как отправную точку для нашего упрощения. Сначала мы включаем каркасный режим так, чтобы вы могли видеть эффект (на вершинах он более нагляден). Добавим следующую строку в функцию SetupCamera:

```
device.RenderState.FillMode = FillMode.WireFrame;
```

Затем, как мы уже решили, Mesh-объект должен быть очищен. Поскольку нам необходимо знать информацию смежности объекта для очистки, мы должны переписать код создания объекта в методе LoadMesh следующим образом:

```
ExtendedMaterial [ ] mtrl;
GraphicsStream adj;
// Load our mesh
mesh = Mesh.FromFile(file, MeshFlags.Managed, device, out adj, out mtrl);
```

Единственным реальным изменением здесь является добавление переменной adj, которая будет возвращать информацию смежности. Затем мы вызываем процедуру FromFile, которая возвращает эти данные. Теперь необходимо применить функцию очистки нашего объекта, для этого в конце функции LoadMesh мы можем добавить следующий код:

```
// Clean our main mesh
Mesh tempMesh = Mesh.Clean(mesh, adj, adj);
// Replace our existing mesh with this one
mesh.Dispose();
mesh = tempMesh;
```

Прежде чем мы изменим наш код, чтобы упростить объект, следует вначале рассмотреть метод упрощения Simplify:

```
public static Microsoft.DirectX.Direct3D.Mesh Simplify (
   Microsoft.DirectX.Direct3D.Mesh mesh ,
   int[] adjacency ,
   Microsoft.DirectX.Direct3D.AttributeWeights vertexAttributeWeights ,
   float [] vertexWeights , System.Int32 minValue ,
   Microsoft.DirectX.Direct3D.MeshFlags options )
```

Структура этого метода должна казаться нам знакомой. Объект Mesh, который мы собираемся упрощать, представляет собой первый параметр, параметр смежности *adjacency* (который может быть определен в виде массива или в виде графического потока) — последующий.

Параметр *AttributeWeights* служит для того, чтобы установить различные коэффициенты, используемые при упрощении объекта. Для большинства приложений следует использовать процедуры, которые не используют это значение, поскольку заданная по умолчанию структура рассматривает только геометрические данные и данные нормали. Лишь в отдельных случаях эти значения должны были бы измениться. Значения по умолчанию для этой структуры, если вы их не пересылаете, имели бы вид:

```
AttributeWeights weights = new AttributeWeights() ;
weights.Position = 1.0f;
weights.Boundary = 1.0f;
weights.Normal = 1.0f;
```

```
weights.Diffuse = 0.0f;
weights.Specular = 0.0f;
weights.Binormal = 0.0f;
weights.Tangent = 0.0f;
weights.TextureCoordinate = new float[]
(0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f };
```

Следующий параметр *vertexWeights* — список весовых коэффициентов для каждой вершины. Если вы пересылаете пустой указатель для этого параметра, принимается что каждый вес будет равен 1.0f.

Далее идет параметр *MinValue*— минимальное число поверхностей или вершин (в зависимости от пересылаеых флажков), до которого вы хотите попробовать упростить объект. Чем меньше это значение, тем менее детальный объект получится. Однако, следует обратить внимание, что выбор этого значения не подразумевает реальное минимальное значение, которое может быть достигнуто. Другими словами, это значение определяет желаемый, а не абсолютный минимум.

Последним параметром *options* для этой функции может быть один из двух флажков. Если вы хотите упростить число вершин, вы должны записать MeshFlags. Simplify Vertex, иначе используйте MeshFlags. SimplifyFace.

Теперь необходимо добавить код, чтобы упростить Mesh-объект, который мы использовали в нашем примере MeshFile, см. главу 5. Желательно иметь перед глазами и первоначальный оригинал объекта (очищенный), и упрощенный, а также иметь возможность переключаться от одного к другому. Таким образом, добавим новое поле, чтобы сохранить упрощенный объект:

```
private Mesh simplifiedMesh = null;
```

Затем необходимо создать упрощенный объект. В конце функции LoadMesh добавьте следующий код:

```
// Get our new simplified mesh
simplifiedMesh = Mesh.Simplify(mesh, adj, null, 1,
   MeshFlags.SimplifyVertex);
Console.WriteLine("Number of vertices in original mesh: {0}",
   mesh.NumberVertices);
Console.WriteLine("Number of vertices in simplified mesh: {0(",
   simplifiedMesh.NumberVertices);
```

Здесь мы попробовали упростить огромный объект до отдельной вершины. В окне вывода следует указать число вершин до и после упрощения. Выполнение приложения на данном этапе не покажет вам упрощенный объект, но выдаст число вершин двух объектов:

```
Number of vertices in original mesh: 4445
Number of vertices in simplified mesh: 391
```

Таким образом, мы упростили наш объект до примерно 8,8 % от его первоначального размера. Для объектов, находящихся на значительном расстоянии от наблюдателя, достаточно трудно найти различие между высоким и низким разрешением этих объектов, кроме того, данный метод позволяет более рационально использовать память и увеличить быстродействие.

Теперь давайте изменять код таким образом, чтобы мы могли видеть результат упрощения на экране с возможностью переключения от одного объекта к другому. Вначале мы должны добавить булеву переменную, чтобы установить текущий объект, который мы отображаем:

```
private bool isSimplified = false;
```

Далее мы должны изменить наш код рендеринга, чтобы отобразить соответствующее подмножество в зависимости от значения нашей булевой переменной. В методе OnPaint перепишите следующий код:

```
device.Material = meshMaterials[i];
device.SetTexture(0, meshTextures[i]);
if (!isSimplified)
{

   mesh.DrawSubset(i);

   }

else

{

   simplifiedMesh.DrawSubset(i);

   }
```

Обратите внимание, что независимо от того, отображаем мы обычный или упрощенный объект, материал и текстура для обоих остаются теми же самыми. Разница лишь в том, какой объект мы вызываем в методе DrawSubset. Теперь осталось предусмотреть возможность переключения нашей логической переменной. Привяжем выполнение этой функции к нажатию клавиши «Пробел», для этого добавьте следующий код:

```
protected override void OnKeyPress(KeyPressEventArgs e)
(
   if (e.KeyChar == '4'
   {
      isSimplified = !isSimplified;
   }
}
```

Теперь, когда мы запускаем это приложение, мы видим наш объект, отображенный в каркасном режиме. Нескольких, имеющихся у этого объекта вершин, вполне достаточно для того, чтобы объект выглядел сплошным. Теперь нажмите клавишу «Пробел» и наблюдайте результат удаления вершин. Эффект весьма значительный, но, когда камера расположена близко к объекту, не совсем лицеприятный. Для управления расстоянием между камерой и объектом добавьте другую булеву переменную:

```
private bool isClose = true;
```

Измените функцию SetupCamera, чтобы иметь возможность отодвигать камеру, в зависимости от значения этой булевой переменной:

```
if (isClose)
{
   device.Transform.View = Matrix.LookAtLH(new Vector3(0,0, 580.0f), new
      Vector3(),new Vector3 (0,1,0));
      }

else
{
   device.Transform.View = Matrix.LookAtLH(new Vector3(0,0, 8580.0f), new
   Vector3(),
      new Vector3(0,1,0));
      }
//device.RenderState.FillMode = FillMode.WireFrame;
```

Легко заметить, что камера отодвинута от объекта на 8000 единиц назад. Также обратите внимание, что здесь мы выключили каркасный режим. Поскольку большинство игр не запускаются в каркасном режиме, ?то даст вам более реалистичное представление. Последняя вещь, которую мы должны сделать, — обеспечить возможность переключать камеру. Для этого мы будем использовать клавишу «М». Добавьте следующие строки в конце метода OnKeyPress:

```
if (e.KeyChar == 'm')

   isClose = !isClose;
```

Запустите приложение еще раз. Пока камера расположена близко, нажмите несколько раз клавишу «пробел». Затем нажмите клавишу «т», чтобы переместить камеру от объекта. Нажмите клавишу «пробел» еще несколько раз и обратите внимание, что теперь эффект гораздо менее драматичен, а результат упрощения при соответствующем положении камеры имеет вполне терпимое качество изображения. На **рис. 7.1** и **рис.7.2** можно сравнить исходный и упрощенный объекты.

Рис. 7.1. Близко расположенный объект с исходным высоким разрешением

Рис. 7.2. Далеко расположенный объект с низким разрешением

Примененная методика может использоваться весьма эффективно, когда необходимо снизить детальное разрешение Mcsh-объекта, и когда нет возможности создать множественные копии одного и того же объекта для разных уровней.

СОХРАНЕНИЕ MESH-ОБЪЕКТОВ

Можно заметить, что выполнение этой операции является весьма трудоемким. Сравните время запуска нашего первоначального приложения MeshFile с новым, описанным только что. Если у вашего художника нет возможности создавать отдельные объекты с низким разрешением, вы наверняка не хотите вынудить пользователя терпеть каждый раз процесс упрощения.

Осталось выбрать золотую середину. Можно выполнять упрощения, используя инструменты, не являющиеся частью основной игры, и сохранять недавно упрощенный объект в свой собственный файл. Это позволит игре по необходимости загружать данные быстро и без повторного выполнения всех процедур упрощения.

Если бы мы захотели сохранить нашу упрощенную версию объекта, мы могли бы добавить код, подобный этому:

```
// Save our simplified mesh for later use
// First we need to generate the adjacency for this mesh
int[] simpleAdj = new int[simplifiedMesh.NumberFaces * 3];
simplifiedMesh.GenerateAdjacency(0.0f, simpleAdj);
using (Mesh cleanedMesh = Mesh.Clean(simplifiedMesh,
            simpleAdj, out simpleAdj))

   cleanedMesh.Save(@"..\..\Asimple.x", simpleAdj,
      mtrl, XFileFormat.Text);
```

Обратите внимание, что мы не получали никакой информации смежности для нашего упрощенного объекта. Поскольку нам эта информация еще понадобится, мы просто сгенерируем эти данные. Необходимо отметить, что после процедуры упрощения необходимо вначале очистить объект и только потом сохранять. Мы выполняем это в операторе «using».

Существует восемь различных похожих перегрузок для метода сохранения Save. Четыре из них сохраняют данные в поток, с которым вы затем оперируете, тогда как четыре других сохраняют данные в файл. Мы выбираем вариант с сохранением в файл. Каждая из перегрузок принимает данные смежности и материалы, используемые для объекта. Информация смежности может быть еще раз определена как массив целых чисел или как графический поток. Половина перегрузок использует структуру EffectInstance, имеющую дело с шейдерными файлами HLSL, которые мы обсудим позже.

Последний параметр функции Save — тип формата файла, который вы хотите сохранить. Существует текстовый, двоичный и сжатый формат. Выберите тот, который вас устраивает больше всего.

Объединение вершин в Mesh-объектах

Другой эффективный способ упрощения объекта состоит в том, чтобы объединить «похожие» или общие вершины. Есть метод в классе mesh, называемый WeldVertices, используемый для объединения общих вершин, имеющих одинаковые атрибуты. Прототип метода включает в себя следующее:

```
public void WeldVertices ( Microsoft.DirectX.Direct3D.WeldEpsilonsFlags
    flags ,
    Microsoft.DirectX.Direct3D.WeldEpsilons   epsilons   ,
    Microsoft.DirectX.Direct3D.GraphicsStream   adjacencyIn   ,
    Microsoft.DirectX.Direct3D.GraphicsStream   adjacencyOut   ,
    out int[] faceRemap ,
    Microsoft.DirectX.Direct3D.GraphicsStream   vertexRemap   )
```

Последние четыре параметра уже подробно обсуждались, поэтому мы не будем останавливаться на них еще раз. Они те же самые, какими были в функции Clean. Первые два параметра контролируют то, как различные вершины объединяются вместе. Первый параметр может иметь одно или несколько значений, приведенных в таблице 7.2:

Таблица 7.2. Флажки для объединяемых вершин

WeldEpsilonsFlags.WeldAll	Объединяет все вершины, которые отмечены в данных смежности как перекрываемые
WeldEpsilonsFlags.WeldPartialMatches	Если данная вершина находится в пределах значения epsilon, заданного структурой WeldEpsilons, измените частично перекрываемую вершину так, чтобы она стала идентичной, и, если все компоненты будут равны, то удалите одну из вершин
WeldEpsilonsFlags.DoNotRemoveVertices	Может использоваться, только если определена структура WeldPartialMatches. Позволяет только изменение вершины, но не удаление
WeldEpsilonsFlags.DoNotSplit	Может использоваться, только если определена структура WeldPartialMatches. Не позволяет разбивать вершину

Структура самого WeldEpsilons очень похожа на структуру AttributeWeights, используемую для упрощения объекта. Единственное различие — есть новый параметр для этой структуры, который управляет фактором мозаичности (tessellation).

Мозаичность объекта — это параметр, характеризующий изменение уровня детальности объекта путем изменения числа треугольников или полигонов или путем разбиения отдельного треугольника на два и более. В дальнейшем, подразумевая мозаичность, мы будем использовать термин «тесселяция».

Чтобы показать эффект использования этого метода, мы еще раз изменим существующий пример MeshFile с целью объединения его вершин. Поскольку в действительности это только один вызов, нет необходимости в составлении громоздкого кода, который мы должны добавить. Мы будем осуществлять объединение вершин при изначально нажатой клавише «Пробел», для этого перепишем метод OnKeyPress следующим образом:

```
protected override void OnKeyPress(KeyPressEventArgs e)
{
   Console.WriteLine("Before: {0}", mesh.NumberVertices);
   mesh.WeldVertices(WeldEpsilonsFlags.WeldAU, new WeldEpsilons(), null,
   null);
   Console.WriteLine("After: {0}", mesh.NumberVertices);
}
```

Теперь нажатие клавиши «Пробел» приведет к следующему текстовому сообщению:

```
Before: 4432
After: 3422
```

В итоге мы получили 91 % вершин от начального количества. Это не 'так впечатляет, как в ранее рассмотренном случае, но зато происходит намного быстрее. Однако, обратили ли вы внимание, как изменились координаты текстуры, когда вы нажимали клавишу «пробел»? Это было вызвано удалением некоторых вершин из общей области. Вблизи это немного заметнее, но на расстоянии на это можно не обращать внимания.

ПРОВЕРКА ДОСТОВЕРНОСТИ ВАШЕГО ОБЪЕКТА

Было бы заманчиво иметь способ, позволяющий определить, действительно ли ваш объект необходимо очистить, или же он уже готов к оптимизации. Естественно, такой способ есть. Класс Mesh имеет метод Validate, который имеет четыре варианта загрузки. Вот один из них:

```
public void Validate ( Microsoft.DirectX.Direct3D.GraphicsStream adjacency ,
    System.String errorsAndWarnings )
```

Есть также перегрузки, которые поддерживают целочисленные массивы для информации смежности, к тому же вы можете использовать функции с параметром ошибок или без него.

Если объект является подходящим, то данный метод будет выполнен успешно, и строка вывода ошибок System.String.Empty (если определена) будет пустой. Если же нет (например, индексы недействительны), то поведение зависит от того, действительно ли строка вывода ошибок определена. Если да, функция возвратит в этой строке сообщение об ошибке, в противном случае, функция пропустит комментарий.

Использование метода Validate перед любым упрощением или оптимизацией помогает устранять отказы, возникающие при выполнении этих методов.

Создание нескольких объектов из одного

Эффективнее, чем просто удаление вершин из вашего объекта, может быть разбиение вашего исходного крупного объекта на различные части. Будем использовать тот же пример MeshFile и попробуем слегка изменить его, разбив на меньшие, но более управляемые объекты.

При разбиении объект преобразуется в массив объектов. В нашем примере мы так же будем держать перед глазами исходный и конечные объекты, чтобы иметь возможность сравнивать их.

Вначале нам необходимо объявить массив для разбиваемых объектов и определить некоторые булевы переменные, чтобы управлять представленными объектами. Добавьте следующие переменные в ваш раздел объявления переменных:

```
// Split information
private Mesh[] meshes = null;
private bool drawSplit = false;
private bool drawAHSplit = false;
private int index = 0;
private int lastlndexTick = System.Environment.TickCount;
```

Эти переменные будут определять созданные в результате разбиения объекты, также как и флажок, указывающий на то, рисуете ли вы исходный объект или отображаете разбиваемый объект (по умолчанию выбирается исходный объект). Если вы рисуете разбиваемый объект, имеется второй флажок, указывающий на то, отображаются ли объекты в массиве или по отдельности (по умолчанию следует выбрать второй вариант).

Необходимо обработать текущий индекс отображаемого объекта (в случае, если отображение выполняется по отдельности) и «мини-таймер» для определения момента переключения к следующему объекту (для того же случая).

Теперь мы можем создавать наш массив объектов. Сразу после вызова LoadMesh в методе InitializeGraphics добавьте следующую секцию кода:

```
meshes = Mesh.Split(mesh, null, 1000, mesh.Options.Value);
```

Мы должны рассмотреть различные параметры функции разбиения. Имеются два варианта загрузки, мы выбираем наиболее полный:

```
public static Microsoft.DirectX.Direct3D.Mesh[ ] Split (Mesh mesh , int[ ]
    adjacencyIn ,
    System.Int32 maxSize , MeshFlags options ,
    out GraphicsStream adjacencyArrayOut ,
    out GraphicsStream faceRemapArrayOut ,
    out GraphicsStream vertRemapArrayOut )
```

Видно, что эта функция принимает в качестве первого параметра объект, который мы хотим разбить. Следующий параметр — *adjacencyIn,* данные о смежности (можно переслать пустой указатель null, поскольку мы уже описывали его). Третий параметр *maxSize* — максимальное число вершин в создаваемых объектах. Для предыдущего примера, максимум составляет 1000 вершин на каждый новый объект. Параметр *options* используется, чтобы определить флажки для каждого из недавно созданных объектов. Последние три параметра возвращают информацию о каждом из новых объектов, в виде потока данных. Мы будем использовать перегрузки без этой информации.

После того как мы создали массив объектов, необходимо задать способ переключения от исходного объекта к полученному массиву. Для этого будем использовать те же самые клавиши, что и ранее: клавиша «Пробел» и «M». Замените метод OnKeyPress следующим кодом:

```
protected override void OnKeyPress(KeyPressEventArgs e)
 {
    if (e.KeyChar == ' ')
    {
        drawSplit = !drawSplit;
    }
    else if (e.KeyChar == V)
    {
        drawAHSplit = IdrawAllSplit;
    }
 }
```

Здесь мы переключаем два режима отображения — отображение исходного и разбиваемого объекта, после чего выбираем отображение разбиваемого объекта целиком или же по отдельности. Теперь мы должны изменить метод DrawMesh, чтобы использовать эти переменные. Замените содержание указанного метода в соответствии с кодом, приведенным в листинге 7.2.

Листинг 7.2. Рендеринг разбиваемых объектов.

```
if ((System.Environment.TickCount - lastlndexTick) > 500)
{
   index++;
   if (index >= meshes.Length)
      index = 0;
   lastlndexTick = System.Environment.TickCount;
}
device.Transform.World = Matrix.RotationYawPitchRoll(yaw, pitch, roll) *
   Matrix.Translation(x, y, z);
for (int i = 0; i < meshMaterials.Length; i++)
{
   device.Material = meshMaterials[i];
   device.SetlexturefO, meshTextures[i]) ;
   if (drawSplit)
   {
      if (drawAllSplit)
      {
         foreach(Mesh m in meshes)
            m.DrawSubset(i);
      }
      else
      {
         meshes[index].DrawSubset(i);
      }
   }
   else
   {
      mesh.DrawSubset (i);
   }
}
```

В этой измененной функции мы определяем приращение индекса с интервалом в пол-секунды для того, чтобы иметь возможность пройти по всей длине нашего массива разбиения. Затем, если мы рисуем разбиваемый объект, мы проверяем, рисуем ли мы весь объект или по отдельности, и поступаем соответствующим образом. В противном случае рисуем исходный объект.

При запуске данного приложения вначале отобразится исходный объект. После нажатия на клавишу «Пробел» начнут отображаться отдельные разбитые объекты. Если вы в этом режиме нажмете клавишу «М», отобразятся все разбитые объекты одновременно. Вы можете сравнить исходный и полученный результат.

Краткие выводы

В этой главе мы узнали все, что нужно знать о стандартном классе Mesh, включая.

- Оптимизацию данных Mesh-объекта.
- Упрощение объектов.

  Создание объектов с новыми компонентами вершинных данных.
- Объединение вершин.

В следующей главе мы исследуем ресурсы Direct3D, затрагивающие глубину отображаемых объектов.

Глава 8. Введение в ресурсы

Ресурсы (Resources) являются основополагающей частью отображения или рендеринга сложных объектов в Direct3D. В этой главе мы рассмотрим более совершенные возможности ресурсов, включая.

- Статические и динамические ресурсы.
  Изменение содержимого буферов, включенных в Mesh-объекты.
  Использование различных флажков блокировки.
  Использование небезопасного режима «unsafe» для максимального выполнения операций.

Класс ресурсов

Класс Resource — относительно маленький класс, но он является базовым для всех ресурсов в Direct3D, так что имеет смысл начать рассмотрение этого раздела книги со списка и назначения каждого метода этого класса. В действительности мы никогда не будем создавать или использовать этот класс непосредственно, так как он является абстрактным, но мы сможем использовать различные методы этого класса, приведенные в таблице 8.1.

Таблица 8.1. Методы Класса «Resource» и параметры

Параметр или метод	Описание
Device	Только для чтения; возвращает устройство для этого ресурса
Type	Только для чтения, возвращает тип ресурса. Подходящие значения для этотого параметра находятся в списке ResourceType
Priority	Чтение — запись, возвращает приоритет этого ресурса. Предполагаются только те ресурсы, которые постоянно находятся в управляемом пуле памяти. Они используются, чтобы определить, когда ресурс может быть удален из памяти. Более низкий приоритет означает более быстрое удаление ресурса из памяти. По умолчанию все управляемые ресурсы имеют приоритет О, а все неуправляемые ресурсы будут всегда иметь приоритет 0. Есть также метод SetPriority, который возвращает старый приоритет после назначения нового
PreLoad	Используйте этот метод для индикации того, что управляемый ресурс может вскоре понадобиться. Это позволит приложению Direct3D перемещать ваш ресурс в видеопамять еще до того, как это будет необходимо. Важно обратить внимание, что если вы уже используете значительное количество видеопамяти, этот метод будет неэффективен

Параметр или метод	Описание
PrivateData members	Имеются три значения, которые позволяют получать и записывать индивидуальные данные для каждого из ресурсов. Эти данные никогда не используются Direct3D и могут использоваться приложением по вашему усмотрению

Как вы можете видеть, класс Resource имеет дело главным образом с расширенными возможностями управляемых ресурсов. Важно обратить внимание на то, что каждый ресурс в Direct3D, также как и пул памяти (определенный в списках Usage и Pool), будет иметь директиву using. Рассмотренные параметры и свойства определяют, каким образом используется ресурс, и где он расположен в памяти (будь это системная память, видеопамять или AGP память).

Ресурсы позволяют центральному процессору обращаться к сохраненным данным обычно с помощью механизма блокировки Locking. В наших примерах до настоящего времени мы формировали наши буферы, каждый раз вызывая метод SetData, и создавали текстуры из файлов. Однако, можно извлекать данные или небольшие подразделы этих данных, блокируя на время эти ресурсы. Мы кратко обсудим эти вопросы в этой главе. Давайте пока рассмотрим первый из ресурсов, которые мы представили.

Использование вершинных и индексных буферов

Вершинные буферы являются основной структурой данных, используемых приложением Direct3D для сохранения данных о вершинах, в то время как индексные буферы играют туже роль, но для индексов. Эти • классы происходят от класса Resource и, таким образом, наследуют методы этого класса, добавляя несколько своих собственных.

Одним из главных параметров класса буферов является описание буфера (description). Структура, возвращаемая из этого раздела, сообщает всю информацию о созданном буфере, включая формат, использование, пул памяти, размер и формат вершины. Возможно, мы бы уже знали эту информацию, если бы создавали буферы самостоятельно, но при создании буфера из внешнего источника это весьма полезная информация.

СТАТИЧЕСКИЕ И ДИНАМИЧЕСКИЕ БУФЕРЫ

Описав два этих метода, связанных с классом буферов, мы можем обсудить различия между статическими и динамическими буферами. Это касается и вершинных, и индексных буферов.

Если буфер был создан с флажком Usage.Dynamic— это динамический буфер, любой другой буфер является статическим. Статические буферы предназначены для элементов, значения которых изменяются не так часто, в то время как динамические буфера оперируют с постоянно изменяющимися данными.

Статический буфер, тем не менее, не подразумевает, что его данные не могут изменяться. Однако, блокировка и обмен информацией со статическим буфером во время его использования может привести к проблемам в процессе выполнения приложения. Прежде чем данные можно будет изменить, процессор должен закончить обработку текущего набора данных, что может потребовать определенного времени. С другой стороны, когда данные в буфере изменяются, процессор ждет окончания этих действий, что также влияет на эффективность не с лучшей стороны.

Если вы предполагаете частое изменение данных, необходимо использовать динамический буфер (флажок Usage.Dynamic), это позволит приложению Direct3D оптимизировать конвейер данных для их частого изменения. Но следует отметить, что использование динамического буфера не даст никаких преимуществ в скорости по сравнению со статическим вариантом.

Нам желательно знать количество ожидаемых элементов в буферах на кадр и соответственно под это выбирать буфер. Для максимального быстродействия рекомендуется иметь один большой статический вершинный буфер для каждого отображаемого формата вершин, а при необходимости возможно использование динамических буферов.

Механизм Locking, блокировка используемых буферов

Механизм блокировки locking кажется одним из наиболее неясных моментов в Direct3D, особенно в Управляемом DirectX. Как выполнять эти процедуры и как увеличивать их быстродействие — достаточно серьезный вопрос для тех, кто хотел бы это использовать.

Прежде чем мы возьмемся за это, выясним что же такое «блокировка» буфера? В действительности это просто действие, разрешающее доступ центрального процессора к разделу данных в вашем ресурсе, при этом ресурс блокируется от каких-либо внешних «вмешательств». Поскольку вы не можете непосредственно «общаться» с графическими устройствами, необходимо найти способ управлять данными вершин из вашего приложения, для этого и нужен механизм блокировки. Рассмотрим различные процедуры или методы блокировки, которые существуют для вершинных и индексных буферов (они принимают те же самые параметры):

```
public System.Array Lock ( System.Int32 offsetToLock ,
   Microsoft.DirectX.Direct3D.LockFlags flags )

public System.Array Lock ( System.Int32 offsetToLock , System.Type
   typeVertex ,
   Microsoft.DirectX.Direct3D.LockFlags flags , params int[] ranJcs )

public Microsoft.DirectX.Direct3D.GraphicsStream Lock ( System.Int32
   offsetToLock ,
   System.Int32 sizeToLock , Microsoft.DirectX.Direct3D.LockFlags flags )
```

Как вы можете видеть, имеется три вида перегрузки, предназначенных для блокировки наши буферов. Начнем с первой, как самой простой. Эта перегрузка доступна, только если вы создавали буфер с помощью конструктора, который принимает описание System.Type и значения вершин или индексов.

Остальные две перегрузки представляются весьма интересными. Первый параметр каждой из них — *offset,* смещение (в байтах), с которого вы хотите запустить механизм блокировки. Если вы желаете блокировать весь буфер, необходимо установить нулевое значение смещения. Обратите внимание, что первые два варианта процедуры возвращают массив данных. Во втором варианте вторым параметром является исходный тип возвращаемого массива. Последний параметр *ranks* используется для определения размера или ранга возвращаемого массива. Для примера, давайте предположим, что у вас есть вершинный буфер (Vector3), включающий в себя координатные данные имеющихся 1200 вершин. Блокировка данного буфера выглядела бы следующим образом:

```
Vector3[] data = (Vector3[])vb.Lock(0, typeof(Vector3), LockFlags.None,
1200);
```

Данный вызов возвратит одномерный массив Vector3s, содержащий 1200 элементов. Теперь, скажем, по некоторым причинам мы захотели бы получить двумерный массив из 600 вершин каждый. Тогда вызов блокировки выглядел бы следующим образом:

```
Vector3[,] data = (Vector3[,])vb.Lock(0, typeof(Vector3), LockFlags.None,
600, 600);
```

Параметр *ranks* является параметром массива и определяет ранг массива, который мы хотели бы возвратить. Используя данный параметр, можно создать и возвратить одномерный, двумерный или трехмерный массив.

БОЛЕЕ ЭФФЕКТИВНАЯ БЛОКИРОВКА БУФЕРОВ

Параметр LockFlags будет задействован при последнем варианте вызова процедуры, но сначала следует сказать несколько слов о недостатках перегрузок с возвращаемыми массивами. Речь идет об эффективности и быстродействии. Предположим, у вас имеется созданный буфер с заданными по умолчанию опциями, и никаких флажков блокировки. При вызове метода блокировки будет происходить следующее.

- Данные вершины блокируются; данные в ячейке памяти сохранены.
- Новый массив подходящего типа распределен, дан размер, указанный в параметре ranks.
- Данные скопированы из блокированной ячейки памяти в наш новый буфер.
- Данный буфер возвращен пользователю, и при необходимости может быть изменен.
- При последующем вызове метода Unlock, уже другая копия данных из массива пересылается назад в блокированную ячейку памяти.
- И окончательно все данные вершины разблокируются.

Не трудно понять, почему этот метод выполняется несколько медленнее, чем мы этого ожидали. Можно удалять только одну из двух копий: определяя при создании буфера флажок «только для записи» Usage.WriteOnly, мы устраняем первую копию, а, определяя при блокировке флажок «только для чтения» LockFlags.Readonly, — вторую.

Последний вариант вызова является наиболее предпочтительным. Он также имеет новый параметр, а именно, размер данных, которые мы хотим блокировать. В других перегрузках это значение вычисляется в течение вызова по формуле sizeof(type)*NumberRanks. Если вы используете последнюю перегрузку, просто пересылайте размер данных, которые вы хотите блокировать (в байтах). Если вы хотите блокировать весь буфер, установите в первых двух параметрах этого метода нулевые указатели.

Эта перегрузка возвратит класс GraphicsStream, который позволит вам непосредственно управлять заблокированными данными без любых дополнительных распределений памяти для массивов и без любых дополнительных копирований в память. Теперь вы можете непосредственно и более эффективно управлять памятью, выигрывая в быстродействии.

Но, тем не менее, не стоит полностью отказываться от операций блокировки с использованием массивов из-за того, что они более медленные. Они достаточно удобны и, если их использовать надлежа-

щим образом, проигрыш в быстродействии по сравнению с операциями блокировки потоковых данных GraphicsStream практически незаметен.

Управление процедурой блокировки

Следует упомянуть о флажках, которые вы можете использовать при блокировке буфера. При блокировке вершинного буфера наиболее подходящими являются следующие флажки:

```
LockFlags.None
LockFlags.Discard
LockFlags.NoOverwrite
LockFlags.NoDirtyUpdate
LockFlags.NoSystemLock
LockFlags.Readonly
```

Если вы не используете флажки блокировки, будет задействован механизм блокировки по умолчанию. Если вы хотите управлять процедурой блокировки, можно использовать флажки, приведенные выше.

Флажок Discard (исключение) может использоваться только для динамических буферов. В этом случае вершинный или индексный буфер будет, и, если существует еще какое-либо подключение к старой памяти, будет возвращен новый блок памяти. Этот флажок весьма полезен, если вы заполняете вершинный буфер динамически. Как только вершинный буфер полностью заполняется, вы блокируете его, используя данный флажок. Эта процедура часто используется в комбинации со следующим типом флажков.

Флажок NoOverwrite (также используется только для динамических буферов) сообщает приложению Direct3D, что вы не будете перезаписывать данные в вершинном или индексном буфере. Использование данного флажка позволит отменить или переопределить вызов и продолжить использовать буфер. Если вы не используете этот флажок, блокировка будет выполняться до тех пор, пока не завершится любой текущий рендеринг. Поскольку при этом вы не можете перезаписывать данные в этот буфер, это будет полезно только при добавлении вершин в буфер.

Стандартное использование этих двух флажков подразумевает использование буфера большого размера, который заполняется динамически. Флажок NoOverwrite используется до тех пор, пока буфер не заполнится полностью. В это время следующая блокировка будет использовать флажок Discard для блокировки буфера. В листинге 8.1. приводится пример кода из процедуры PointSprites (точечные спрайты), поставляемой вместе с DirectX SDK.

Листинг 8.1. Выборка из метода PointSprites.

```
if (++numParticlesToRender == flush)
{
   // Done filling this chunk of the vertex buffer. Lets unlock and
   // draw this portion so we can begin filling the next chunk.
   vertexBuffer.Unlock();
   dev.DrawPrimitives(PrimitiveType.PointList, baseParticle,
            numParticlesToRender);
   // Lock the next chunk of the vertex buffer. If we are at the
   // end of the vertex buffer, LockFlags.Discard the vertex buffer and start
   // at the beginning. Otherwise, specify LockFlags.NoOverWrite, so we can
   // continue filling the VB while the previous chunk is drawing.
   baseParticle += flush;
   if (baseParticle >= discard)
      baseParticle = 0;
   vertices = (PointVertex[])vertexBuffer.Lock(baseParticle *
      DXHelp.GetTypeSize(typeof(PointVertex)), typeof(PointVertex),
      (baseParticle != 0) ? LockFlags.NoOverwrite : LockFlags.Discard,
      flush);
   count = 0;
   numParticlesToRender = 0;
}
```

В этой выборке регистрируется момент, когда данные полностью накопились и готовы к отображению на экране (вызов DrawPrimitives). Если такой момент наступил, мы разблокируем наш буфер и производим рендеринг. Затем мы определяем, действительно ли мы находимся в конце нашего буфера (baseParticle > = discard), и блокируем наш буфер еще раз. Мы запускаем блокировку в конце буфера, пересылая флажок NoOverwrite, за исключением случая, когда baseParticle имеет ненулевое значение, тогда мы блокируем данные в начале буфера, используя флажок Discard. Это позволяет нам продолжить добавление новых точечных спрайтов к нашей сцене, пока буфер не заполнится, после чего мы исключим наш старый буфер и начнем заполнять новый.

Рассмотрев эти два типа флажков, мы можем передти к оставшимся. Самый простой из них — Readonly, который, судя по его названию, сообщает Direct3D, что вы не будете записывать данные в буфер. Когда мы имеем дело с перегрузками блокировки, которые возвращают массивы, данная операция имеет некоторые преимущества, поскольку не происходит копирования данных, обновляемых при вызове разблокировки Unlock.

Флажок NoDirtyUpdate предотвращает любое изменение ресурса, находящегося в недействительной области. Без этого флажка блокировка ресурса автоматически приведет к добавлению недействительной области к этому ресурсу.

Последний из флажков — NoSystemLock, как правило, используется нечасто. Обычно, когда вы блокируете ресурс в видеопамяти, резервируется критическая секция «systemwide» в рамках всей системы, не позволяя каким-либо образом изменять режимы визуального отображения до тех пор, пока действует блокировка. Использование флажка NoSystemLock отключает это свойство. Данное действие может быть полезно только для операций блокировки, занимающих весьма непродолжительное время, и только если вы все еще нуждаетесь в увеличении скорости выполнения. Таким образом, часто использовать данный флажок не рекомендуется.

Естественно, после вызова блокировки и после выполнения необходимых операций мы должны разблокировать буфер. Для этого используется механизм разблокировки. У этой функции нет никаких параметров, разблокировка должна вызываться каждый раз, когда используется блокировка и наоборот, невыполнение этого требования может вызвать ошибки при процедуре рендеринга.

Использование текстурных ресурсов

Все ресурсы текстур в Управляемом DirectX происходят от отдельного класса BaseTexture, принадлежащего общему классу Resource. Существует несколько новых методов, основанных на использовании объектов класса BaseTexture, см. таблицу 8.2.

Таблица 8.2. Методы класса BaseTexture и параметры

Параметр или метод	Описание
AutoGenerateFilterType	Параметр чтения-записи, описывает, каким образом mipmap подуровни сгенерированы автоматически. Он принимает значение списка TextureFilter (фильтра текстур), и автоматически воссоздает любые mipmap подуровни, если тип фильтра изменен. Заданный по умолчанию тип фильтра для генерации mipmap уровня — TextureFilter.Linear (линейная фильтрация), в случае если драйвер не поддерживает этот режим, будет использоваться TextureFilter.Point (точечная фильтрация). Если устройство не поддерживает выбранный тип фильтра, процедура фильтрации игнорируется. Все параметры фильтра являются используемыми, за исключением значения «None». Можно видеть, что значение TextureFilterCaps структуры Capabilities (Возможности) определяет, какие типы фильтров поддерживаются вашим устройством
LevelOfDetail	Параметр чтения-записи, определяет наиболее детальный (в плане разрешения) уровень управляемых текстур

Параметр или метод	Описание
GenerateMipSubLevels	Этот метод автоматически генерирует подуровни текстуры mipmap, о которой мы только что упомянули. Он будет использовать следующее свойство, чтобы определить режим фильтрации и использовать его при создании этих уровней
LevelCount	Только для чтения, определяет число уровней используемой текстуры. Например, текстура mipmapped (текстура с разрешением, изменяющимся по мере удаления отображаемого объекта, см. ниже в «Структура Mipmaps»), имела бы число уровней, определяемое значением LevelCount

СТРУКТУРА MIPMAPS

Проще говоря, mipmap представляет собой цепочку или последовательность текстур, первая из текстур имеет самое большое значение детального разрешения. Значение разрешения каждой последующей текстуры уменьшается в два раза от значения предыдущей. Например, при значении разрешения первой текстуры 256x256, следующий уровень будет иметь 128x128, затем 64x64 и так далее, по мере удаления объекта от наблюдателя. Цепочки Mipmap используются приложением Direct3D для управления качеством отображаемых текстур, что требует значительного места в памяти. Данный подход позволяет существенно экономить ресурсы памяти при сохранении приемлемого разрешения в целом.

При создании текстуры один из используемых параметров определяет число уровней, которые вы хотите иметь в этой текстуре. Это число уровней связано непосредственно с цепочкой текстур, которые были описаны выше. При задании нулевого значения для этого параметра приложение Direct3D автоматически создаст набор mipmaps от вашей первоначальной текстуры с максимальной разрешающей способностью до последней, имеющей разрешение 1x1. В нашем примере, используя «0» для этого параметра, и имея первоначальное разрешение 256x256, будет создана цепочка из девяти текстур: 256x256, 128x128, 64x64, 32x32, 16x16, 8x8, 4x4, 2x2, и 1x1.

При вызове функции SetTexture приложение Direct3D будет автоматически проводить фильтрацию различных текстурируемых цепочек mipmaps, основываясь на текущей установке свойств MipFilter в классе Sampler. Если вспомнить, в нашей игре «Dodger», которую мы написали в главе 6, мы устанавливали фильтры растяжения и сжатия для наших текстур дороги. Для фильтров mipmap используются те же подходы.

Блокировка текстур и получение описаний

Подобно вершинным и индексным буферам, текстуры также имеют механизм блокировки. Дополнительно к известным нам методам здесь появляются две новые особенности, которые не присущи ресурсам геометрии (вершины, нормали и пр.), а именно: недействительные области (dirty regions) и фоновые объекты (backing object). Для начала изучим механизм блокировки для текстур.

Блокировка текстур сходна с подобной операцией над нашими «геометрическими» буферами с двумя имеющимися различиями. Во-первых, вызов блокировки принимает параметр уровня level множественной текстуры, по отношению к которому мы хотим выполнить блокировку. Во-вторых, вы можете заблокировать отдельно полигон (к примеру, куб для объемных текстур), или же блокировать всю поверхность текстуры, используя при этом перегрузку без определения этого параметра.

При работе с кубическими текстурами, необходим еще один дополнительный параметр — сторона отображаемого куба, которую вы хотите заблокировать. В качестве этого параметра вы можете использовать значение из списка CubeMapFace.

Обратите внимание, что некоторые из процедур блокировки полигонов LockRectangle могут возвращать параметр шага (pitch). Это дает возможность узнать о количестве данных в каждом ряду (в байтах).

В то время как блокировки текстуры могут принимать параметр «type» для блокировки определенных типов данных, сами текстуры непосредственно сохраняют только информацию о цветах. Если вы желаете использовать одну из перегрузок, использующих тип, вы должны убедиться, что данный тип имеет тот же самый формат, что и формат пиксела текстуры.

Теперь мы рассмотрим блокировку текстуры на примере. Будем использовать знакомый нам MeshFile. Мы удалим текстуру, которая используется для рендеринга, и заменим ее на другую, созданную динамически для нашего приложения. Нам потребуется два режима для этой текстуры: тот, который циклически повторяет текстуры, и тот, который является абсолютно случайным. Для этого нам понадобятся следующие переменные:

```
private uint texColor = 0;
private bool randomTextureColor = false;
```

Мы планируем использовать 32-разрядный формат пиксела для нашей текстуры, таким образом, мы сохраняем наш цвет также в 32-разрядном формате («uint», целочисленный). С помощью булевой переменной мы выбираем, либо мы создаем случайные цвета текстуры, либо ис-

пользуем циклически повторяющуюся текстуру. Теперь мы должны создать метод, который будет выполнять блокировку текстуры и ее обновление или изменение. Чтобы показать различия между методами с использованием массивов (array methods) и небезопасными методами (unsafe methods), рассмотрим оба случая. Метод с использованием массива приведен в листинге 8.2.

Листинг 8.2. Заполнение текстуры с использованием массива.

```
private void FillTexture(Texture t, bool random)
{
   SurfaceDescription s = t.GetLevelDescription(0);
   Random r = new Random() ;
   uint[,j data = (uint[,])t.LockRectangle(typeof(uint), 0,
      LockFlags.None, s.Width, s.Height);
   for (int i = 0; i < s.Width; i++)
   (
      for (int j = 0; j < s.Height; j++)
      (
         if (random)
         (
            data[i,j] = (uint)Color.FromArgb(
               r.Next(byte.MaxValue),
               r.Next(byte.MaxValue),
               r.Next (byte.MaxValue)) .ToArgb() ;
         }
         else
         {
            data[i,j] = texColor++;
            if (texColor >= 0x00ffffff)
               texColor = 0;



   t.OnlockRectangle(0);
}
```

Здесь мы берем существующую текстуру и сначала получаем ее описание, чтобы узнать и задать размер данных для заполнения. Зная ширину и высоту данных, мы блокируем данные в двухмерном массиве (ширина/высота). При блокировке данного массива используется целое число без знака (поскольку используется 32-разрядный формат пиксела), при этом информация о флажках блокировки не важна. Так как мы создаем только один уровень mipmap последовательности, мы определяем значение для блокировки нулевого уровня в текстуре.

После того, как текстура блокирована, перебираются значения возвращенного массива текстуры, ее цвета изменяются соответственно либо случайным образом, либо с помощью приращения (цвет нашей циклической текстуры). И, наконец, после того как эти данные полностью изменены, мы разблокируем нашу текстуру.

Рассмотрим, что же происходит при использовании небезопасного метода, вариант которого приведен в листинге 8.3.

Листинг 8.3. Заполнение текстуры с использованием небезопасного метода.

```
private unsafe void FillTextureUnsafe(Texture t, bool random)
{
   SurfaceDescription s = t.GetLevelDescription(0);
   Random r = new Random() ;
   uint* pData = (uint*)t.LockRectangle(0,
      LockFlags.None).IntemalData.ToPointer() ;
   for (int i = 0; i < s.Width; i++)
   {
      for (int j = 0; j < s.Height; j++)
      {
         if (random)
         {
            *pData = (uint)Color.FromArgb(
               r.Next(byte.MaxValue),
               r.Next(byte.MaxValue),
               r.Next(byte.MaxValue)).ToArgb ();
         }
         else
         {
            *pData = texColor++;
            if (texColor >= 0x00ffffff)
               texColor = 0;
         }
         pData++;
         }
         }
   t.UnlockRectangle(0);
}
```

Обратите внимание, что единственные отличия этой функции — задание небезопасного режима и то, как мы изменяем наши данные. Мы блокируем буфер данных потока и используем значение IntemalData (как указатель IntPtr), чтобы получить доступ к данным, которые мы будем изменять. Мы управляем данными непосредственно через указатель Pointer, и в конце операции вновь разблокируем нашу текстуру.

Чтобы компилировать этот метод, вы должны включить в свойствах проекта возможность компиляции небезопасного кода. Для этого в разделе Configuration Properties, установите значение «true» для свойства Allow Unsafe Code Blocks.

Теперь, когда мы имеем функцию, позволяющую заполнить текстуру различными цветами, необходимо изменить функцию LoadMesh, чтобы использовать данную функцию, а не пытаться создавать текстуры через файл. Замените код создания материала в методе LoadMesh следующим кодом:

```
// Store each material and texture
for (int i = 0; i < mtrl.Length; i++)
{
   meshMaterials[i] = mtrl[i].Material3D;
   meshTextures[i] = new Texture(device,
             256, 256, 1, 0, Format.X8R8G8B8, Pool.Managed);
#if (UNSAFE)
   FillTextureUnsafe(meshTextures [i], randomTextureColor);
#else
   FillTexture(meshTextures [i], randomTextureColor);
#endif
}
```

Как вы можете видеть, мы больше не пытаемся загружать текстуру из файла. Для каждой текстуры в нашем массиве мы создаем новую текстуру, используя формат пиксела X8R8G8B8 (32-разрядный формат). Мы создаем текстуру 256x256 с одним только уровнем. Затем мы заполняем нашу текстуру, используя только что описанный метод и оператор #if для выбора. Вы можете так же определить выбор Unsafe в вашем файле кода и использовать его вместо предложенного варианта.

ВРАЩЕНИЕ С ПРИВЯЗКОЙ К ЧАСТОТЕ СМЕНЫ КАДРОВ

Следует упомянуть один момент, на котором мы уже останавливались раньше. Необходимо вспомнить то, как случайно изменялись цвета текстуры и скорость вращения модели при нажатии на клавишу. Это связано с тем, что движение модели в этом случае зависит от скорости смены кадров в большей степени, чем от системного таймера.

Имея вновь созданную с помощью новой функции текстуру, вы можете выполнять приложение и создавать более красочные модели. Для обновления текстуры необходимо вызывать метод заполнения текстуры

каждый раз при смене кадра. Кроме того, мы можем устанавливать просмотр нашей текстуры. Перепишите метод SetupCamera, см. листинг 8.4, вставив туда нашу функцию.

Листинг 8.4. Установка опций камеры.

```
private void SetupCamera()
{
   device.Transform.Projection = Matrix.PerspectiveFovLH((float)Math.PI / 4,
      this.Width / this.Height, 1.0f, 10000.0f);
   device.Transform.View = Matrix.LookAtLH(new Vector3(0,0, 580.0f),
      new Vector3(), new Vector3(0,1,0));
   device.RenderState.Lighting = false;
   if (device.DeviceCaps.TextureFilterCaps.SupportsMinifyAnisotropic)
      device.SamplerState[0].MinFilter = TextureFilter.Anisotropic;
   else if (device.DeviceCaps.TextureFilterCaps.SupportsMinifyLinear)
      device.SamplerState[0].MinFilter = TextureFilter.Linear;
   if (device.DeviceCaps.TextureFilterCaps.SupportsMagnifyAnisotropic)
      device.SamplerState[0].MagFilter = TextureFilter.Anisotropic;
   else if (device.DeviceCaps.TextureFilterCaps.SupportsMagnifyLinear)
      device.SamplerState[0].MagFilter = TextureFilter.Linear;
   foreach(Texture t in meshTextures)
#if (UNSAFE)
   FillTextureUnsafe(t, randomTextureColor);
#else
   FillTexture(t, randomTextureColor);
#endif
}
```

Мы выключили освещение для того, чтобы видеть разнообразие наших цветов без какого-либо эффекта затенения, и включили текстурный фильтр (если он поддерживается). Последнее, что необходимо сделать, — каким-то образом включить случайные цвета текстуры. Будем использовать нажатие любой клавиши в качестве переключателя между двумя режимами текстурирования. Для этого добавьте следующее:

```
protected override void OnKeyPress(KeyPressEventArgs e)
{
   randomTextureColor = ! randomTextureColor;
}
```

Использование случайного подбора цветов создает эффект анимации. Раскраска объекта на экране напоминает белый шум или картинку в телевизоре при отсутствии принимаемого сигнала, см. **рис.8.1.**

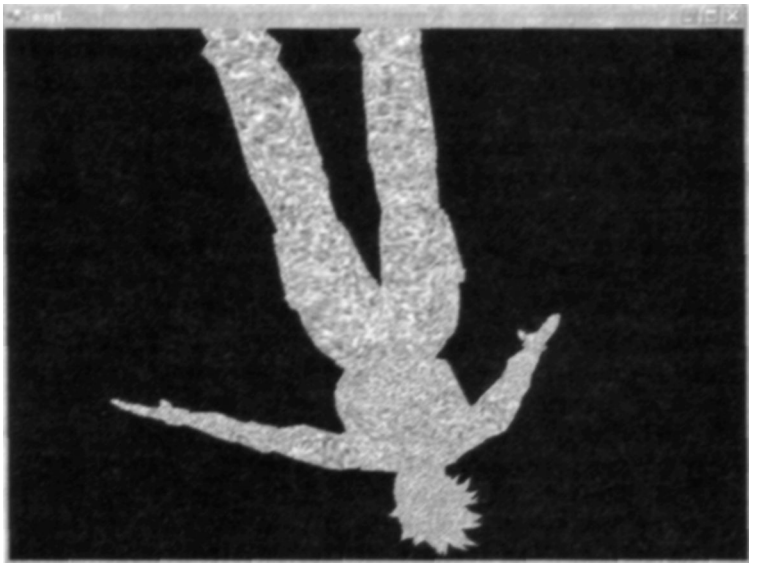


Рис. 8.1 «Снежная» текстура

Краткие выводы

В этой главе мы обсудили различные ресурсы, доступные в приложении Direct3D и их использование, включая.

- Статические и динамические ресурсы.
- Изменение содержимого буферов, включенных в объекты Mesh.
- Использование различных флажков блокировки.
- Использование небезопасного режима для увеличения быстродействия.

В нашей следующей главе мы начнем рассматривать более сложные Mesh-типы, доступные для нашего использования.

Глава 9. Применение других типов Mesh

Упрощение Mesh-объектов

Итак, мы ознакомились с основным классом Mesh, включенным в библиотеку расширений. Однако, существуют три других типа Mesh, которые мы еще не рассматривали, и которые изучим в этой главе.

Мы уже знакомы по главе 7 с созданием упрощенных объектов Mesh с помощью методов Simplify. Поскольку общий случай использования упрощенных объектов дает самое низкое разрешение модели, сохранять множественные версии объектов — это, вероятно, не самая хорошая идея, особенно если вы даже не собираетесь использовать некоторых из них.

Для более эффективного упрощения объектов можно использовать объект SimplificationMesh. Однако, непосредственно при использовании данного подхода мы не можем выполнять рендеринг, необходимо сначала получить реальный объект из упрощенного. Если вспомнить, раньше при упрощении объектов мы использовали два «режима» для нашего примера. Первый режим отображал неупрощенный исходный объект с близкого расстояния. Второй вариант позволял отобразить менее детальный объект (за счет сокращения числа вершин и сторон) с большего расстояния. На данном этапе мы будем использовать нечто похожее, более того, мы будем использовать подход для рендеринга загружаемых из файла объектов, рассмотренный в главе 5. Сейчас мы просто добавим наш код к рассмотренному примеру. Первое, что нам будет необходимо, — объявление для нашего объекта функции SimplificationMesh, а также переменной для управления местоположением камеры. Добавьте следующие переменные:

```
private SimplificationMesh simplifiedMesh = null;
private float cameraPos = 580.0f;
```

Переменная, описывающая глубину размещения камеры, позволит легко изменять ее местоположение. Мы также должны преобразовать координаты для правильного размещения камеры в сцене. В методе SetupCamera перепишите процедуру преобразования следующим образом:

```
device.Transform.View = Matrix.LookAtLH(new Vector3(0,0, cameraPos),
   new Vector3(), new Vector3(0,1,0));
```

Очевидно, что, поскольку метод SetupCamera вызывается каждый раз при смене кадра, необходимо также каждый раз переписывать перемен-

ную местоположения камеры. Эта переменная находится там же, где мы записываем данные при упрощении объекта. Следует обратить внимание на то, что мы не имеем никаких дополнительных mesh-объектов. Каждый раз, упрощая наш объект, мы будем просто заменять его новым.

Необходимо переписать метод LoadMesh, чтобы удостовериться, что наш объект очищен, и упрощенный объект создан должным образом. Мы должны заменить этот метод на приведенный в листинге 9.1:

Листинг 9.1. Создание упрощенного объекта Mesh.

```
private void LoadMesh(string file)
(
   ExtendedMaterial[] mtrl;
   GraphicsStream adj ;
   // Load our mesh
   mesh = Mesh.FromFile(file, MeshFlags.Managed, device, out adj, out mtrl!
   // If we have any materials, store them
   if ((mtrl != null) && (mtrl.Length > 0))
   {
      meshMaterials = new Material[mtrl.Length];
      meshTextures = new Texture[mtrl.Length];
      // Store each material and texture
      for (int i = 0; i < mtrl.Length; i++)
      {
         meshMaterials[i] = mtrl[i] .Materia3D;
         if ((mtrl[i].TextureFilename != null) &&
             (mtrl[i].TextureFilename != string.Empty))
         {
            // We have a texture, try to load it
            meshTextures[i] = TextureLoader.FromFile(device, @"..\..\" +
               mtrl[i].TextureFilename);
         }
      }
   }
   // Clean our main mesh
   Mesh tempMesh = Mesh.Clean(mesh, adj, adj);
   // Replace our existing mesh with this one
   mesh.Dispose();
   mesh = tempMesh;
   // Create our simplification mesh
   simplifiedMesh = new SimplificationMesh(mesh, adj);
}
```

Как видите, мы должны имееть информацию смежности и для очищенного, и для упрощенного объектов. После загрузки объекта и создания текстуры, мы очистим наш объект, перед тем как упростить его.

Практически, это все, что было необходимо выполнить. Осталось только добавить код, позволяющий нам перемещать объект дальше от камеры, и уменьшать детальное разрешение объекта. Расстоянием до камеры мы будем управлять с помощью клавиатуры, см. листинг 9.2:

Листинг 9.2. Обработчик событий для операции KeyPress.

```
protected override void OnKeyPress(KeyPressEventArgs e)
(
   if (e.KeyChar == '+')
   {
      cameraPos += (MoveAmount * 2);
      simplifiedMesh.ReduceFaces(mesh.NumberFaces - MoveAmount);
      simplifiedMesh.ReduceVertices(mesh.NumberVertices - MoveAmount);
      mesh.Disposed;
      mesh = simplifiedMesh.Clone(simplifiedMesh.Options.Value,
      simplifiedMesh.VertexFormat, device);
   }
   if (e.KeyChar == 'W')
      device.RenderState.FillMode = FillMode.WireFrame;
   if (e.KeyChar == 'S')
      device.RenderState.FillMode = FillMode.Solid;
}
```

Обратите внимание, что здесь мы используем неопределенную константу, чтобы управлять необходимым перемещением, активизируемым с помощью нажатия клавиши. Можно определить константу следующим образом:

```
private const int MoveAmount = 100;
```

В описываемом методе нажатие клавиши «W» переключает выполнение приложения в каркасный режим WireFrame, который достаточно нагляден для того, чтобы «разглядеть» полигоны на отображаемом объекте. Нажатие клавиши «S» переключит нас обратно к сплошному отображению объекта. При нажатии клавиши «+» мы будем отодвигать камеру от нашей модели до определенного значения. Удаляя объект от камеры, мы уменьшаем число сторон и вершин в соответствии с указанной константой MoveAmount. Для того чтобы отобразить наш упрощенный объект, необходимо использовать его в методе рендеринга вместо первоначального объекта.

Выполняя приложение и нажимая клавиши «+» и «W» (соответственно приближая или удаляя объект и переключая отображение от каркасного исполнения к сплошному), мы практически не увидим разницы между отображаемыми объектами. Теперь можно добавить сопровождающий текст к нашей сцене, чтобы показать число сторон и вершин отображаемого в данный момент объекта. Для управления шрифтом добавьте следующую переменную:

private Microsoft.DirectX.Direct3D.Font font = null;

Мы также должны инициализировать наш шрифт прежде, чем начать отображать его. После вызова LoadMesh в методе IntializeGraphics добавьте код инициализации:

```
// Create our font
font = new Microsoft.DirectX.Direct3D.Font(device, new System.Drawing.Font
   "Arial", 14.0f, FontStyle.Bold | FontStyle.Italic));
```

Изменяя и и параметры можно выбирать различный тип, размер н стиль шрифта. И, наконец, осталось отобразить лот текст. После вызова DrawMesh можно добавить следующий код:

```
font.DrawText(null, string.Format("Number vertices in mesh:
   mesh.numberVertices), new Rectangle (10, 10, 0, 0),
   DrawTextFormat.NoClip, Color.BlanchedAlmond);

font.DrawText(null, string.FormatCНивЬег faces in mesh: (0)*,
   mesh.HumberFaces), new Rectangle(10, 30, 0, 0),
   DrawTextFormat.UoClip, Color.BlanchedAlmond);
```

Отображая комментарии к нашим объектам и оперируя указанными клавишами, легко заметить, что при удалении объекта от камеры число вершин и рисуемых поверхностей уменьшается, рис.9.1.

Рис. 9.1 Упрощенный Mesh-объект

Одна из основных проблем, с которой придется столкнуться в данном приложении, заключается в том, что при обратном или повторном приближении камеры к объекту мы не можем восстановить потерянные в процессе упрощения вершины. Для данного случая нет никакого метода восстановления после использования, к примеру, метода упрощения ReduceVertices.

Объекты и функции упрощения SimplificationMesh разработаны и предназначены только для того, чтобы упростить объект, без обратного его восстановления.

Для решения этой проблемы существуют более совершенные объекты Mesh.

Управление степенью детализации, класс прогрессивных Meshes-объектов

Бывают случаи, когда можно просто упростить объект, без его восстановления (например, модель удаляющейся после выстрела ракеты). Это предполагает, что после упрощения данного объекта мы не сможем улучшить или вернуть его качество. Для примера с ракетой в этом нет необходимости. Но данный пример является далеко не общим случаем. Как правило, возникает необходимость, когда нам необходимо уменьшить и затем поднять степень детализации объекта. Этого можно достичь, используя более совершенные методы, к которым можно отнести прогрессивные meshes-объекты.

Чтобы показать поведение прогрессивных mesh-объектов, мы напишем приложение, опираясь на те методы, которые уже использовали для упрощения объектов, например SimplificationMesh. Однако, вместо того, чтобы только упрощать объект, мы также попробуем поднимать уровень детализации при обратном приближении камеры к объекту. Как и раньше, мы начнем с примера, использующего загрузку объекта из 'файла.

Класс ProgressiveMesh происходит от общего класса mesh-объектов BaseMesh. Вы можете использовать класс ProgressiveMesh, объявив его предварительно для вашего объекта:

```
private ProgressiveMesh progressiveMesh = null;
```

Необходимо также заменить переменную старого класса Mesh на новую переменную progressiveMesh. Соответственно, для корректной компиляции с новой переменной необходимо переписать метод LoadMesh.

Мы будем использовать подобный метод, в конце которого сгенерируем прогрессивный mesh-объект. Используйте код, приведенный в листинге 9.3.

Листинг 9.3. Загрузка объекта Progressive Mesh

```
private void LoadMesh(string file)
{
   ExtendedMaterial[] mtrl;
   GraphicsStream adj;
   // Load our mesh
   using(Mesh mesh = Mesh.FromFile(file, MeshFlags.Managed, device,
                out adj, out mtrl))
   {
      // If we have any materials, store them
      if ((mtrl != null) && (mtrl.Length > 0))
      {
         meshMaterials = new Material[mtrl.Length] ;
         meshTextures = new Texture[mtrl.Length];
         // Store each material and texture
         for (int i = 0; i < mtrl.Length; i++)
         {
            meshMaterials[i] = mtrl[i].Material3D;
            if ((mtrl[i].TextureFilename != null) &&
                (mtrl[i].TextureFilename != string.Empty))
            {
               // We have a texture, try to load it
               meshTextures[i] = TextureLoader.FromFile(device,
                  @"..\..\" + mtrl[i].TextureFilename);
            }
         }
      }
      // Clean our main mesh
      using(Mesh tempMesh = Mesh.Clean(mesh, adj, adj))
      {
         // Create our progressive mesh
         progressiveMesh = new ProgressiveMesh(tempMesh, adj,
            null, 1, MeshFlags.SimplifyVertex);
         // Set the initial mesh to the max
         progressiveMesh.NumberFaces = progressiveMesh.MaxFaces;
         progressiveMesh.NumberVertices = progressiveMesh.MaxVertices;
      }
   }
}
```

Обратите внимание, что для создания прогрессивного mesh-объекта мы используем два временных объекта: очищенный объект и параметр, необходимый для конструктора прогрессивного mesh-объекта — минимальное число вершин или сторон, которые мы хотим получить в создаваемом объекте, в зависимости от пересылаемого значения MeshFlags (либо упрощение поверхностей Simplify Face, либо упрощение вершин Simplify Vertex). Ее-

тественно, это только приближенный подход, но, тем не менее, мы можем получить некоторые преимущества от его использования.

Также обратите внимание, что в данном случае мы сразу задаем число вершин и/или сторон в максимальные значения (максимальный уровень детализации), что позволит изменять текущий уровень детализации, используемый при отображении mesh-объекта.

Для компиляции приложения необходимо переписать вызов DrawSubset в методе DrawMesh следующим образом:

```
progressiveMesh.DrawSubset(i);
```

Запуская приложение, обратите внимание, что оно ведет себя точно так же, как и при использовании первоначального варианта. Наши модели вращаются и выглядят полностью детализированными. Теперь мы должны обработать нашу привычную операцию нажатия клавиш и движение камеры, используя подход, аналогичный примененному ранее. Необходимо также объявить переменную местоположения камеры и константу перемещения:

```
private float cameraPos = 580.0f;
private const int MoveAmount = 100;
```

Еще раз необходимо привести в соответствие преобразования координат, чтобы каждый раз обновлять местоположение нашей камеры. Замените код преобразования координат на следующий:

```
device.Transform.View = Matrix.LookAtLH(new Vector3(0,0, cameraPos),
   new Vector3(), new Vector3(0,1,0));
```

Окончательно, мы должны как обычно обработать операции с клавишами, см. листинг 9.4:

Листинг 9.4. Обработчик событий для операции KeyPress.

```
protected override void OnKeyPress(KeyPressEventArgs e)
(
   if (e.KeyChar == '+')
   {
      cameraPos += (MoveAmount * 2);
      progressiveMesh.NumberVertices =
          ((BaseMesh)progressiveMesh).NumberVertices - MoveAmount;
      progressiveMesh.NumberFaces =
          ((BaseMesh)progressiveMesh).NumberFaces - MoveAmount;
   }
```

```
    if (e.KeyChar == '-')
    {
       cameraPos -= (MoveAmount * 2);
       progressiveMesh.NumberVertices =
           ((BaseMesh)progressiveMesh).NumberVertices + MoveAmount;
       progressiveMesh.NumberFaces =
           ((BaseMesh)progressiveMesh).NumberFaces + MoveAmount;
    }
       if (e.KeyChar == 'W')
           device.RenderState.FillMode = FillMode.WireFrame;
       if (e.KeyChar == 'S')
           device.RenderState.FillMode = FillMode.Solid;
}
```

Опять мы выполняем переключение между каркасным и сплошным режимом отображения, используя клавиши «W» или «S». Для уменьшения уровня детализации и соответствующего перемещения камеры от объекта используется клавиша «+». Обратите внимание, что, прежде чем получить число сторон и вершин, мы должны определить наш объект progressiveMash в классе BaseMesh, чтобы согласовать установку и получение параметров.

Легко заметить, что для увеличения уровня детализации и соответствующего приближения камеры к объекту используется клавиша «-». Таким образом, при приближении камеры к объекту увеличивается число отображаемых вершин и сторон.

СОХРАНЕНИЕ МНОЖЕСТВЕННЫХ УРОВНЕЙ ДЕТАЛИЗАЦИИ

Обычно сохранение множественных объектов с различной степенью детализации проще, чем наличие одного большого прогрессивного mesh-объекта с возможностью управления степенью детализации. Вы можете ознакомиться с прогрессивным mesh-объектом, который включен в поставляемый DirectX SDK, и можете применить методы TrimByFaces и TrimByVertices, чтобы изменять степень детализации объекта.

Теперь было бы неплохо добавить к приложению некоторый текст с комментариями о числе отображаемых вершин и сторон. Не будем повторяться с инициализацией и определением шрифта, поскольку данная процедура была подробно описана в этой же главе. Здесь можно лишь добавить следующее:

```
font.DrawText(null, string.Format("Number vertices in mesh: {O}",
    ((BaseMesh)progressiveMesh) .NumberVertices), new Rectangle(l0, 10, 0, 0),
    DrawTextFormat.NoClip, Color.BlanchedAlmond);
```

```
font.DrawText(null, string.Format("Number faces in mesh: {0}",
    ((BaseMesh)progressiveMesh).NumberFaces), new Rectangle(10, 30, 0, 0),
    DrawTextFormat.NoClip, Color.BlanchedAlmond) ;
```

Рендеринг патч-объектов. Тесселяция объектов

Упрощение объектов достаточно распространенная операция на практике, но что если ваш объект уже итак слишком прост? С одной стороны, вы имеете достаточную пропускную способность, чтобы отобразить даже большее количество вершин в вашем объекте, чем вы имеете. Но с другой стороны, вы не можете себе позволить увеличить детализацию уже имеющейся модели, это ничего не даст в плане информационного разрешения. Проблему можно решить, применяя патч-объекты. Большинство современных графических 3D-программ моделирования ориентированы на применение патч-типов mesh или некоторых примитивов старшего разряда, таких, как NURBS (неравномерный рациональный би-сплайн для описания кривых поверхностей) или секционированные/разделенные поверхности.

В данной книге мы рассмотрим лишь описание применения патч-объектов, в частности, для увеличения уровня детализации модели. Для этого мы создадим небольшое приложение, позволяющее нам улучшить качество разрешения нашего загружаемого объекта.

Снова мы загружаем наш пример из файла (глава 5). Также нам понадобится модель для нашего примера, которая находится на CD диске. Программа, находящаяся на CD диске, использует две модели, включенные в DirectX SDK (tiger.x — тигр и cube.x — куб), плюс, маленькую модель сферы, которую вы найдете там же. Убедитесь, что вы копируете текстуру для модели tiger.bmp. Далее нам нужно добавить следующие переменные:

```
private float tessLevel = 1.0f;
private const float tesslncrement = 1.0f;
private string filename = @"..\..\sphere.x";
private Microsoft.DirectX.Direct3D.Font font = null;
```

Таким образом, сейчас мы имеем текущий уровень тесселяции (tessLevel), характеризующий степень мозаичности объекта, который первоначально установлен в значение l.Of, а также приращение этого уровня (1.Of). Пока мы зафиксировали эти значения, но на практике можно смело изменять эти параметры. Далее мы должны сохранить название текущей модели, которую мы загружаем, а также шрифт для сопровождающего текста.

Нам также понадобится внести поправки в метод SetupCamera, на этот раз мы будем использовать несколько меньшие значения удаления камеры от объекта, поскольку имеем меньшие размеры объекта.

В соответствии с вышесказанным, запишите метод следующим образом:

```
private void SetupCamera()
{
   device.Transform.Projection = Matrix.PerspectiveFovLH(
      (float)Math.PI / 4, this.Width / this.Height, 1.0f, 100.0f);
   device.Transform.View = Matrix.LookAtLH(new Vector3(0,0, 5.0f),
      new Vector3(), new Vector3(0,1,0));
   device.Lights[0].Type = LightType.Directional;
   device.Lights[0].Diffuse = Color.DarkKhaki;
   device.Lights[0].Direction = new Vector3(0, 0, -1);
   device.Lights[0].Commit();
   device.Lights[0].Enabled = true;
}
```

Мы просто устанавливаем камеру и освещение. Не очень эффективно вызывать это каждый раз при смене кадра, но для ясности, мы будем делать это в данном примере.

Конечно, необходимо проверить, поддерживает ли наша графическая плата направленное освещение или нет, но для простоты предположим, что поддерживает.

Вместо использования отдельного вызова LoadMesh, как мы делали прежде, мы должны добавить следующие секции кода:

```
// Create our patch mesh
CreatePatchMesh(filename, tessLevel);

// Create our font
font = new Microsoft.DirectX.Direct3D.Font(device, new System.Drawing.Font
   ("Arial", 14.0f, FontStyle.Bold | FontStyle.Italic));

// Default to wireframe mode first
device.RenderState.FillMode = FillMode.WireFrame;
```

Очевидно, что здесь мы создаем патч-объект вместе с нашим шрифтом.

Далее мы определяем по умолчанию каркасный режим заполнения, поскольку так намного проще видеть новые треугольники или полигоны.

Однако, мы пока не имеем метода для создания патч-объектов, так что следует добавить такой код, см. листинг 9.5:

Листинг 9.5. Создание патч-объекта mesh.

```
private void CreatePatchMesh(string file, float tessLevel)
{
   if (tessLevel < 1.0f) // Nothing to do
      return;
   if (mesh != null)
      mesh.Disposed;
   using (Mesh tempmesh = LoadMesh(file))
   {
      using (PatchMesh patch = PatchMesh.CreateNPatchMesh(tempmesh))
      {
         // Calculate the new number of faces/vertices
         int numberFaces = (int)(tempmesh.NumberFaces
            * Math.Pow(tessLevel, 3));
         int numberVerts = (int)(tempmesh.NumberVertices
            * Math.Pow(tessLevel, 3));
         mesh = new Mesh(numberFaces, numberVerts, MeshFlags.Managed
               MeshFlags.Use32Bit, tempmesh.VertexFormat, device);
         // Tessellate the patched mesh
         patch.Tessellate(tessLevel, mesh);
      }
   }
}
```

Если уровень тесселяции, который мы в настоящее время используем, меньше чем 1.0f (по умолчанию), ничего в этом методе делать не нужно, и процедура прекращает выполнение. В противном случае мы должны создать новый mesh-объект и применить к нему процедуру тесселляции, сохранив его затем вместо старого. Изменим наш текущий метод LoadMesh, который не возвращает значения данных нормалей, на следующий, см. листинг 9.6:

Листинг 9.6. Загрузка mesh-объекта с данными нормалей.

```
private Mesh LoadMesh(string file)
{
   ExtendedMaterial[] mtrl;
   // Load our mesh
   Mesh mesh = Mesh.FromFile(file, MeshFlags.Managed, device,
               out mtrl);
   // If we have any materials, store them
   if ((mtrl != null) && (mtrl.Length > 0))
   {
      meshMaterials = new Material[mtrl.Length];
```

```
        meshTextures - new Texture[mtrl.Length];
        // Store each material and texture
        for (int i = 0; i < mtrl.Length; i++)
        {
           meshMaterials[i] = mtrl[i].Material3D;
           if ((mtrl[i].TextureFilename != null) &&
               (mtrl[i].TextureFilename != string.Empty))

              // We have a texture, try to load it
              meshTextures[i] = TextureLoader.FromFile(device,
                 @"..\..\" т mtrl[i].TextureFilename);
           }
        }
     }
     if ((mesh.VertexFormat & VertexFormats.Normal) != VertexFormats.Normal)
     {
        // We must have normals for our patch meshes
        Mesh tempMesh = mesh.Clone(mesh.Options.Value,
           mesh.VertexFormat | VertexFormats.Normal, device);
        tempMesh.ComputeNormals();
        mesh.Dispose();
        mesh = tempMesh;
     }
     return mesh;
  }
```

Здесь нет ничего такого, чего мы не видели прежде. Если в нашем объекте нет никаких нормалей, мы имитируем новый объект и вычисляем нормали, поскольку они потребуются для тесселяции нашего патч-объекта. И, наконец, мы возвращаем созданный объект (с нормалями).

ДИРЕКТИВА USING

Обратите внимание, что и для возвращенного объекта, и для патч-объекта, которые мы создаем в этом методе, мы используем ключевое слово «using». Эта директива предполагает автоматическое размещение объекта в классе, если он оставляет область действия оператора.

Затем мы создаем новый патч-объект на базе возвращенного объекта. Поскольку мы будем создавать новый патч, подразумевая изменение уровня тесселяции объекта, нам необходимо знать число новых вершин и сторон в этом объекте. После того как мы рассчитали и создали новый объект, мы, наконец, можем его тессилировать. Обратите внимание на использование флажка MeshFlags.Use32Bit. Поскольку после тесселяции мы мо-

жем получить возвращенный объект очень большого размера, необходимо вначале убедиться, что мы можем поддерживать такие объекты.

Мы почти готовы к запуску этого приложения. Как и ранее, добавим текст к отображаемой сцене. После вызова метода рисования нашего объекта добавьте следующее:

```
font.DrawText(null, string.Format
    ("Number Vertices: {0}\r\nNumber Faces: {1}",
   mesh.NumberVertices, mesh.NumberFaces),
   new Rectangle(10,10,0,0),
   DrawTextFormat.NoClip, Color.Black);
```

Осталось написать сам метод тесселяции объекта. Мы будем использовать код, приведенный в листинге 9.7:

Листинг 9.7. Обработчик событий операции KeyPress для тесселяции объекта.

```
protected override void OnKeyPress(KeyPressEventArgs e)

    if (e.KeyChar == '+')
    {
       tessLevel += tesslncrement;
       CreatePatchMesh(filename, tessLevel);
    }
    if (e.KeyChar == '-')
    {
       tessLevel -= tesslncrement;
       CreatePatchMesh(filename, tessLevel);
    }
    if (e.KeyChar == 'c')
    {
       filename = @"..\..\cube.x";
       tessLevel = 1.0f;
       CreatePatchMesh(filename, tessLevel);
    }
    if (e.KeyChar == 'o')
    {
       filename = @"..\..\sphere.x";
       tessLevel = l.0f;
       CreatePatchMesh(filename, tessLevel);
    }
    if (e.KeyChar == 't')
    {
       filename = @". ,\. \tiger.x";
       tessLevel = l.0f;
```

```
            CreatePatchMesh(filename, tessLevel);
        }
        if (e.KeyChar == 'W')
            device.RenderState.FillMode = FillMode.WireFrame;
        if (e.KeyChar == 's')
            device.RenderState.FillMode = FillMode.Solid;
    }
```

Как и раньше, для переключения каркасного и сплошного режимов отображения используются клавиши «W» или «S», для изменения уровня тесселяции клавиши «*» и «-».

Клавиша «С» используется для того, чтобы отобразить объект «куб», клавиша «Т» — объект «тиф», а клавиша «О» возвращает нас к изображению сферы.

Примеры тесселированных объектов

На рисунках 9.2 — 9.5 приведены примеры отображаемых объектов с использованием различных вариантов тесселяции.

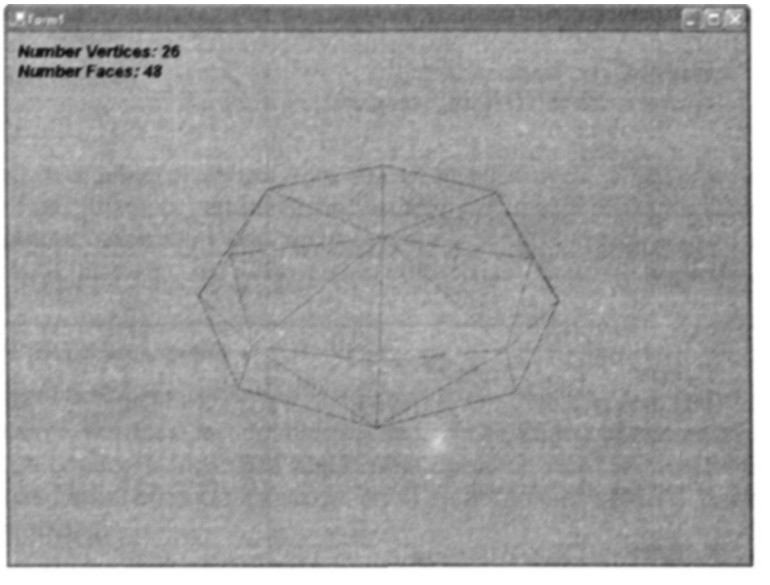


Рис. 9.2. Изображение сферы с маленьким разрешением

Изображение сферы (рис.9.2). построенной с помощью треугольников, выглядит не так реалистично.

Однако, если мы разбиваем каждый треугольник на несколько, качество изображения улучшается (рис.9.3)

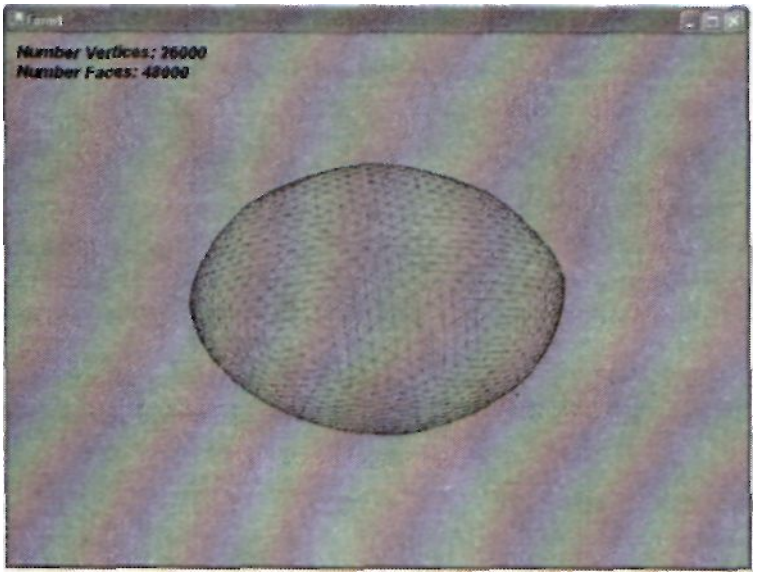


Рис. 9.3. Изображение сферы с улучшенным разрешением

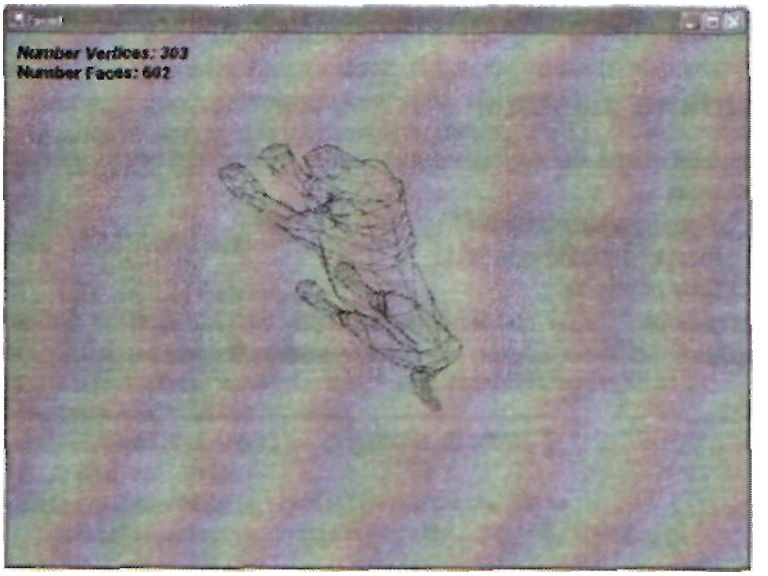


Рис. 9.4. Изображение тигра с нормальным разрешением

Как можно видеть на **рис.9.4,** изображение тигра выглядит несколько мозаично, хотя использование текстуры определенно помогает скрашивать этот недостаток **(рис.9.5).**

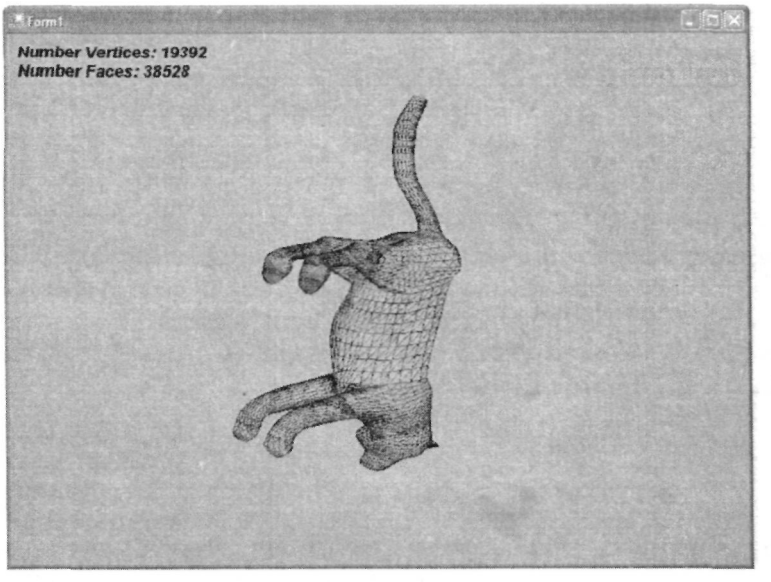


Рис. 9.5. Изображение тигра с высоким разрешением

Увеличение параметра тесселяции позволяет значительно улучшить качество картинки.

Краткие выводы

В данной главе мы ознакомились с дополнительными типами объектов, включая.

- Упрощенные объекты.
  Прогрессивные объекты.
- Патч-объекты.

В следующей главе мы изучим дополнительные классы библиотеки расширений, включая линии, карты окружающей среды и рендеринг текстур.

Глава 10. Использование вспомогательных классов

Рисование линий

Прорисовка линий была рассмотрена в главе 4, когда мы имели дело с примитивами типа набора линий или полосок, используемых при рендеринге. Однако, при использовании таких примитивов мы не можем изменять ширину линий, кроме того, объекты, выполненные таким образом, оказываются не достаточно сглаженными.

Класс Line может использоваться в зависимости от типа используемого приложения и является достаточно простым и удобным. Рассмотрим пример, в котором попробуем отобразить случайное число линий, непрерывно рисуемых на экране.

Для этого необходимо создать и подготовить новый проект Direct3D, добавить необходимые ссылки к сборкам, включая класс и using-директиву для списка имен. Так же понадобится создать индивидуальную переменную для устройства и установить форму окна рендеринга. Эти действия мы уже неоднократно проделывали и раньше, поэтому не будем на этом останавливаться.

Линии могут быть нарисованы либо в аппаратных координатах устройства, либо в мировых координатах. В нашем примере мы рассмотрим только экранное пространство, кроме того, мы задействуем буфер глубины. Используйте метод, приведенный в листинге 10.1, чтобы инициализировать графику.

Листинг 10.1. Инициализация устройства.

```
  public void InitializeGraphics()
{
      // Set our presentation parameters
      PresentParameters presentParams = new PresentParameters();
      presentParams.Windowed = true;
      presentParams.SwapEffect = SwapEffect.Discard;
      // Create our device
      device = new Device(0, DeviceType.Hardware, this,
         CreateFlags.SoftwareVertexProcessing, presentParams);
   }
```

Приведенный код представляет собой ту же конструкцию, которую мы использовали и раньше, но только без буфера глубины. Функция рендеринга в данном случае будет использовать метод OnPaint точно так же, как и в предыдущих примерах:

```
protected override void OnPaint(System.Windows.Forms.PaintEventArgs e)
{
    device.Clear(ClearFlags.Target, Color.Black, 1.0f, 0);
    device.BeginScene();
    //Draw some lines
    DrawRandomLines();
    device.EndScene();
    device.Present();
    // Let them show for a few seconds
    System.Threading.Thread.Sleep(250);
    this.Invalidated;
}
```

На данном этапе мы очищаем наше устройство, создаем сцену, вызываем процедуру рисования нескольких линий, завершаем сцену и отображаем ее. Также мы устанавливаем задержку длительностью в четверть секунды, для того чтобы видеть отображаемые линии в течение нескольких циклов. Теперь необходимо вызвать метод рисования линий, см. листинг 10.2.

Листинг 10.2. Рисование произвольно появляющихся линий.

```
private void DrawRandomLines()
{
    Random r = new Random();
    int numberLines = r.Next(50);
    using (Line l = new Line(device))
    {
        for (int i = 0; i < numberLines; i++)
        {
            int numVectors = 0;
            while (numVectors < 2)
                numVectors = r.Next(4);
            Vector2[] vecs = new Vector2[numVectors];
            for(int inner = 0; inner < vecs.Length; inner++)
            {
                vecs[inner] = new Vector2(r.Next(this.Width),
                    r.Next(this.Height));
            }
            // Color
            Color c = Color.FromArgb(r.Next(byte.MaxValue),
                r.Next(byte.MaxValue), r.Next(byte.MaxValue));
            int width = 0;
            while (width == 0)
            width = r.Next(this.Width / 100);
            // Set the width
```

```
            l.Width = width;
            // Should they be antialiased?
            l.Antialias = r.Next(50) > 25 ? true : false;
            // Draw the line
            l.Begin();
            l.Draw(vecs, c);
            l.End();
        }
    }
}
```

Каждый раз при вызове метода сначала определяется случайным образом число линий для рисования, максимум 50 линий. Затем создается отдельный объект line, для прорисовки каждой линии. Возможно также создание одного общего объекта для всех линий.

Затем случайным образом выбирается число точек в линии, минимум две, после чего выбирается случайное положение для этих точек (начальной и конечной) с учетом ширины и высоты нашего окна. Также выбирается случайный цвет и ширина линии (также с учетом размеров окна). Режим сглаживания выбирается случайным образом. В зависимости от истинного или ложного значения логической переменной сглаживания, линии будут отображаться соответственно либо «зубчатыми», либо более сглаженными.

Следует обратить внимание на связку begin/draw/end в листинге 10.2, определяющую рисование линии. Существуют и другие варианты рисования линии, например, булева переменная GILines, которая определяет, должны ли использоваться линии стиля OpenGL или нет (для выбора по умолчанию устанавливается значение «false»). Можно также использовать метод DrawTransform для рисования линий в трехмерном пространстве, а не в используемых здесь аппаратных координатах.

Итак, теперь необходимо переписать основной метод следующим образом:

```
static void Main()
{
    using (Form1 frm = new Form1()
    {
        // Show our form and initialize our graphics engine
        frm.Show();
        frm.InitializeGraphics();
        Application.Run(frm);
    }
}
```

При выполнении этого приложения экран заполняется набором беспорядочно расположенных линий различных размеров, как гладких, так и «зазубренных». Количество линий на экране будет изменяться случайным образом, как на рис.10.1.

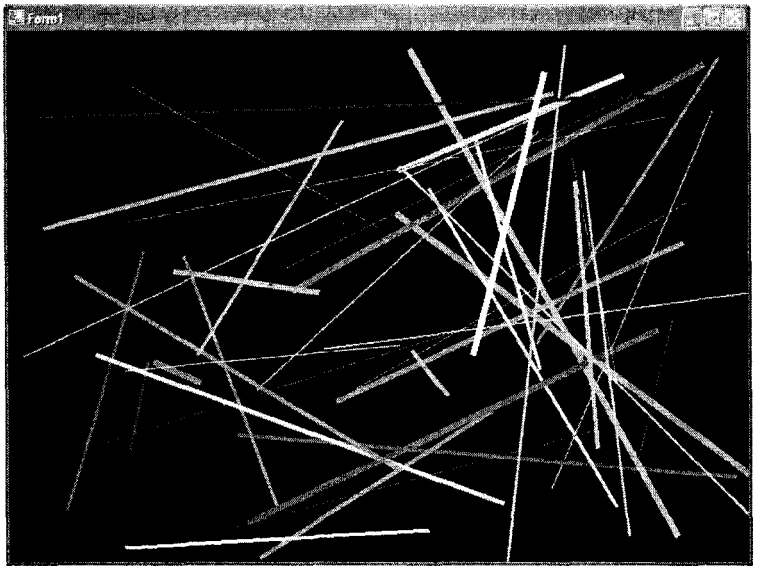


Рис. 10.1. Изображение случайных линий на экране

Отображение текста

Мы уже обсуждали некоторые вопросы, касающиеся отображения текста на экране, однако, это было довольно поверхностное изучение. Теперь рассмотрим эти вопросы более подробно. Напомним, в предыдущих примерах для объекта шрифта Font имелись два пространства имен Microsoft.DirectX.Direct3D и System.Drawing, которые необходимо различать.

Как и раньше, мы можем переименовать используемый класс в соответствии со следующей директивой «using»:

```
using Direct3D = Microsoft.DirectX.Direct3D;
```

Теперь можно создать новый проект для примера отображения текста, предварительно проверив добавление необходимых переменных, ссылок, операторов «using» и пр. Необходимо также проверить наличие индивидуальной переменной нашего устройства и привязку окна для рендеринга. После этого необходимо добавить следующие переменные:

```
private Direct3D.Font font = null;
private Mesh mesh = null;
private Material meshMaterial;
private float angle = O.Of;
```

Здесь мы объявляем шрифт и материалы отображаемого текста. В качестве объекта выбирается несложный тесненный текст. Угловой параметр (angle parameter), который мы использовали и ранее, позволит трехмерному тексту вращаться со скоростью, зависящей от скорости смены кадров. Теперь необходимо инициализировать графику, используя метод, приведенный в листинге 10.3.

Листинг 10.3. Инициализация графики для рисования фрагмента текста.

```
public void InitializeGraphicsf)
i
   // Set our presentation parameters
   PresentParameters presentPararas = new PresentParametersO;
   presentParams.Windowed = true;
   presentParams.SwapEffect = SwapEffeet.Discard;
   presentParams.AutoDepthStencilFormat = DepthFormat.D16;
   presentParams.EnableAutoDepthStencil = true;
   // Create our device
   device = new Device(0, DeviceType.Hardware, this,
      CreateFlags.SoftwareVertexProcessing, presentParams);
   device.DeviceReset += new System.EventHandler(this.OnDeviceReset);
   OnDeviceReset(device, null);
   // What font do we want to use?
   System.Drawing.Font localFont = new System.Drawing.Font
      ("Arial", 14.Of, FontStyle.Italic);
   // Create an extruded version of this font
   mesh = Mesh.TextFromFont(device, localFont, "Managed DirectX",
         O.OOlf, 0.4f);
   // Create a material for our text mesh
   meshMaterial = new Material!);
   meshMaterial.Diffuse = Color.Peru;
   // Create a font we can draw with
   font = new Direct3D.Font(device, localFont);
```

Таким образом, мы создаем аппаратное устройство с буфером глубины, и отслеживаем событие сброса устройства DeviceReset. Поскольку нам необходимо устанавливать освещение и камеру каждый раз после сброса устройства, мы привяжем выполнение этого кода к обработчику событий. Затем с помощью System.Drawing.Font мы задаем шрифт, кото-

рый будем использовать как основной при рисовании двух- и трехмерных фрагментов тестов. Для данного примера выбран шрифт Arial 14, хотя это не принципиально.

Итак, сначала мы создаем трехмерный объект текста в виде тесненной надписи «Managed DirectX», причем создаем отдельный mesh для каждой отображаемой строки. Затем мы определяем материал и цвет отображаемого 2D-шрифта.

РЕНДЕРИНГ ОБЪЕМНОГО ТЕСНЕННОГО ТЕКСТА

Рисунок предварительно выбранного двухмерного текста может быть выполнен с помощью двух простых треугольников, тогда как при отображении трехмерного тесненного текста могут понадобиться тысячи треугольников.

Как уже упоминалось, для привязки камеры и освещения мы используем обработчик событий, для этого добавляем следующий код:

```
private void OnDeviceReset(object sender, EventArgs e)
{
    Device dev = (Device)sender;
    dev.Transform.Projection = Matrix.PerspectiveFovLH(
        (float)Math.PI / 4, this.Width / this.Height, 1.0f, 100.0f);
    dev.Transform.View = Matrix.LookAtLH(new Vector3(0,0, -9.0f),
        new Vector3(), new Vector3(0,1,0));
    dev.Lights[0].Type = LightType.Directional;
    dev.Lights[0].Diffuse = Color.White;
    dev.Lights[0].Direction = new Vector3(0, 0, 1);
    dev.Lights[0].Commit();
    dev.Lights[0].Enabled = true;
}
```

Камера и освещение выбираются только для трехмерного тесненного текста. Для двухмерного текста, который предварительно преобразован и освещен, эти опции не востребованы.

Теперь добавим метод, который будет рисовать трехмерный тесненный текст:

```
private void Draw3DText(Vector3 axis, Vector3 location)
{
    device.Transform.World = Matrix.RotationAxis(
        axis, angle) *
        Matrix.Translation(location);
    device.Material = meshMaterial;
    mesh.DrawSubset(0);
```

```
    angle += O.Olf;
}
```

Как вы можете видеть, мы пересылаем координаты mesh-объекта в мировом пространстве и оси, относительно которых будет вращаться текст. Данный метод рисования напоминает метод DrawMesh, который мы использовали в предыдущих примерах, он просто задает материал и рисует первое подмножество тесненного объекта (для этого объекта это будет только одно подмножество). Далее мы увеличиваем угловой параметр. В нашем случае при управлении «анимацией» мы будем привязывать скорость перемещения объекта к частоте смены кадров (как мы уже выясняли, это не совсем обосновано, но пока остановимся на этом). Теперь добавим метод рисования 2D-текста:

```
private void Draw2DText (string text, int x, int y, Color c)
 {
    font.DrawText(null, text, new Rectangle(x, y,
       this.Width , this.Height ),
       DrawTextFormat.NoClip | DrawTextFormat.ExpandTabs !
       DrawTextFormat.WordBreak , c);
 }
```

Здесь мы пересылаем текст, который хотим отобразить, и месторасположение (в аппаратных координатах устройства) верхнего левого угла текста. Последний параметр, естественно, — цвет. Обратите внимание на то, что в нашем запросе к DrawText мы включаем ширину и высоту окна отображения, а также пересылаем флажок WordBreak (перенос строки). Все это позволяет управлять переходом текста на новую строку или в новое положение, учитывая размер соответствующего ограничивающего прямоугольника.

Теперь мы можем выполнить процедуру рендеринга текстового фрагмента, для этого необходимо переписать наш метод OnPaint, см. листинг 10.4.

Листинг 10.4. Рендеринг фрагмента текста.

```
protected override void OnPaint(System.Windows.Forms.PaintEventArgs e)
 {
    device.Clear(ClearFlags.Target | ClearFlags.ZBuffer, Color.Black, l.Of, 0);
    device.BeginScene();
    Draw2DText("Here's some text", 10, 10, Color.WhiteSmoke);
    Draw2DText("Here's some text\r\nwith\r\nhard\r\nline breaks",
       100, 80, Color.Violet);
    Draw2DText("This\tis\tsome\ttext\twith\ttabs.",
```

```
        this.Width / 2, this.Height - 80 , Color.RoyalBlue);
    Draw2DText("If you type enough words in a single sentence, " +
        "you may notice that the text begins to wrap. Try resizing " +
        "the window to notice how the text changes as you size it.",
        this.Width / 2 + this.Width / 4, this.Height / 4 , Color.Yellow);
    // Draw our two spinning meshes
    Draw3DText(new Vector3(1.0f, 1.0f, 0.0f),
        new Vector3(-3.0f, 0.0f, 0.0f));
    Draw3DText(new Vector3(0.0f, 1.0f, 1.0f),
        new Vector3(0.0f, -1.0f, 1.0f)) ;
    device.EndScene();
    device.Present ();
    this.Invalidated;
}
```

Обратите внимание, что сначала мы рисуем простой текст, расположенный в верхнем левом углу экрана. Затем мы рисуем новый текст с перенесенными строками. Далее мы рисуем некоторые слова, отделенные табуляцией (вместо пробелов). Поскольку мы включили флажок ExpandTabs, они будут раздвинуты при рендеринге. После всего этого мы рисуем одно большое предложение, в котором нет установленного нами переноса строки:

```
static void Main()
{
    using (Form1 frm = new Form1()
    {
        // Show our form and initialize our graphics engine
        frm.Show() ;
        frm.InitializeGraphics() ;
        Application.Run(frm);
    }
}
```

РАСШИРЕННЫЕ ВОЗМОЖНОСТИ ШРИФТА

Класс Font рисует текст, основываясь на текстурах, которые создаются в этом классе. Рендеринг этих символов выполняется через графический интерфейс GDI и может быть на самом деле достаточно медленным. Возможно, было бы целесообразнее вызвать два метода предварительной загрузки класса Font в течение запуска, например: PreloadCharacters, чтобы загрузить определенный набор символов, или PreloadText, чтобы загрузить определенную строку.

При запуске приложения на экране отобразится текст, который можно увидеть на **рис. 10.2.** Размеры окна, как и размер текстов, можно изменять.

Рис. 10.2. Отображаемый текст

Рендеринг на поверхности

Если вы когда-нибудь запускали гоночный симулятор, то наверняка помните о возможности использования зеркала заднего обзора. Эти эффекты можно получить, отображая ту же самую сцену (несколько раз с различной привязкой камеры) на имеющуюся текстуру. На первый взгляд это кажется довольно сложным, но на самом деле это вполне реализуемо. Вернемся к примеру из главы 5 и попробуем изменить его должным образом. Естественно, в начале нам необходимо объявить новую текстуру, которую мы будем отображать, плюс, некоторые параметры, которые следует добавить сразу:

```
private Texture renderTexture = null;
private Surface renderSurface = null;
private RenderToSurface rts = null;
private const int RenderSurfaceSize = 128;
```

Мы объявили текстуру и поверхность, на которую мы будем отображать эту текстуру, а также вспомогательный класс для рендеринга поверхности. Мы также объявили размер создаваемых поверхностей. В методе InitializeGraphics мы используем для создания текстур и поверхностей знакомый нам обработчик события сброса. Замените вызов конструктора устройства следующим кодом:

```
device = new Device(0, DeviceType.Hardware, this,
        CreateFlags.SoftwareVertexProcessing, presentParams);
device.DeviceReset +=new EventHandler(OnDeviceReset);
OnDeviceReset(device, null);
```

Задание параметров и поддерживаемых устройством опций приведено в листинге 10.5.

Листинг 10.5. Сброс устройства или установка параметров по умолчанию

```
private void OnDeviceReset(object sender, EventArgs e)
{
   Device dev = (Device)sender;
   if (dev.DeviceCaps.VertexProcessingCaps.SupportsDirectionalLights)
   f
      uint maxLights = (uint)dev.DeviceCaps.MaxActiveLights;
      if (maxLights > 0)
      {
         dev.Lights[0].Type = LightType.Directional;
         dev.Lights[0].Diffuse = Color.White;
         dev.Lights[0].Direction = new Vector3(0, -1, -1);
         dev.Lights[0].Commit();
         dev.Lights[0].Enabled = true;
      }
      if (maxLights > 1)
      {
         dev.Lights[1].Type = LightType.Directional;
         dev.Lights[1].Diffuse = Color.White;
         dev.Lights[1].Direction = new Vector3(0, -1, 1);
         dev.Lights[1].Commit();
         dev.Lights[1].Enabled = true;
      }
}
rts = new RenderToSurface(dev, RenderSurfaceSize, RenderSurfaceSize,
   Format.X8R8G8B8, true, DepthFormat.D16);
   renderTexture = new Texture(dev, RenderSurfaceSize, RenderSurfaceSize, 1,
      Usage.RenderTarget, Format.X8R8G8B8, Pool.Default);
   renderSurface = renderTexture.GetSurfaceLevel(0);
}
```

На данном этапе мы проверяем, поддерживаются ли в устройстве те или иные возможности. Для начала мы выясняем, поддерживает ли устройство направленное освещение. Если да, мы включаем его, полагая, что устройство поддерживает несколько источников света. В нашем случае будем использовать два достаточно ярких источника направленного

освещения, первый -— для освещения передней стороны нашей модели, второй — для освещения объекта сзади.

После создания и включения освещения создается вспомогательный класс RenderToSurface, использующий константы размеров поверхности. Обратите внимание, что многие из параметров для этого конструктора соответствуют параметрам, которые можно найти в представлении устройства. В данном примере используются общие значения, и было бы полезно использовать те же значения из существующей структуры представления параметров.

Итак, мы создаем нашу текстуру, определяем объект рендеринга и выбираем тот же формат, что и для вспомогательного класса. Все отображаемые текстуры объекта должны находиться в сброшенном пуле памяти. Получая поверхность из текстуры, мы должны также сохранить и ее.

Поскольку мы привязали установку освещения к обработчику события сброса, необходимо удалить данную установку из метода SetupCamera, уже существующего в этом примере. Далее нужно добавить новый метод для рендеринга объекта на поверхность, см. метод, приведенный в листинге 10.6.

Листинг 10.6. Рендеринг на поверхность.

```
private void RenderlntoSurface()
f
   // Render to this surface
   Viewport view = new Viewport();
   view.Width = RenderSurfaceSize;
   view.Height = RenderSurfaceSize;
   view.MaxZ = 1.0f;
   rts.BeginScene(renderSurface, view);
   device.Clear(ClearFlags.Target ! ClearFlags.ZBuffer, Color.DarkBlue, 1.0f,
   Ob-
   device.Transform.Projection = Matrix.PerspectiveFovLH((float)Math.PI / 4,
      this.Width / this.Height, 1.0f, 10000.0f);
   device.Transform.View = Matrix.LookAtLH(new Vector3(0,0, -580.0f),
      new Vector3(), new Vector3(0, 1,0));
   DrawMesh(angle / (float)Math.PI, angle / (float)Math.PI * 2.0f,
      angle / (float)Math.PI / 4.0f, 0.0f, 0.0f, 0.0f);
   DrawMesh(angle / (float(Math.PI, angle / (float)Math.PI / 2.0f,
      angle / (float)Math.PI * 4.0f, 150.0f, -100.0f, 175.0f);
   rts.EndScene(Filter.None);
)
```

Данный код представляет собой обычный метод рендеринга. Мы вызываем сцену, устанавливаем камеру и рисуем наш объект. Затем мы вы-

зываем текстуры для нашей сцены и отображаем каждый элемент сцены. В данном случае, помимо обычного расположения камеры, мы еще располагаем камеру и с другой стороны нашей модели, чтобы мы могли видеть, что происходит, если б находились «сзади» модели. Обратите также внимание, что в этой сцене у нас имеется два варианта отображения: первый, когда мы видим только одну модель (представление спереди, по умолчанию), и второй, когда мы добавляем изображение модели с расположенной позади объекта камеры (в этом случае мы будем видеть обе модели).

Параметром процедуры BeginScene является поверхность, на которую будет осуществлен рендеринг объекта. Поскольку мы используем поверхность, восстановленную или полученную из текстуры, любые обновления этой поверхности будут отражаться и на текстуре. Метод EndScene позволяет использовать текстурную фильтрацию для нашей поверхности. В нашем примере мы пока не знаем ничего о поддержке устройством данного фильтра, поэтому фильтрацию опускаем. Последнее, на что следует обратить внимание, — мы изменили цвет для текстурируемой сцены, для того чтобы показать различие между «реальной» сценой и той, «другой» сценой.

Таким образом, мы должны слегка изменить наш метод рендеринга. Для начала необходимо отобразить сцену в нашу текстуру прежде, чем мы начнем отображать ее в нашем основном окне. Добавьте вызов к нашему методу в качестве первого пункта в методе OnPaint:

```
RenderlntoSurface();
```

Далее мы хотим разместить текстуру на экране. В нашем примере мы можем использовать один из вспомогательных классов, называемый Sprite (более подробно мы его обсудим главой позже). Данный класс позволяет простым образом отображать текстуру в аппаратных координатах устройства. Добавьте следующую секцию кода перед методом завершения сцены EndScene:

```
using (Sprite s = new Sprite(device))
{
   s.Begin(SpriteFlags.None);
   s.Draw(renderTexture, new Rectangle(O, 0, RenderSurfaceSize,
      RenderSurfaceSize), new Vector3(0, 0, 0), new Vector3(0, 0, l.Of),
      Color.White);
   s.End();
}
```

Этот код просто отображает поверхность всей текстуры в левом верхнем углу экрана. Результат рендеринга можно увидеть на **рис. 10.3.**

Рис. 10.3. Рендеринг на поверхность

Рендеринг текстур Environment Maps

Environment Maps — карты окружающей среды, использующиеся для моделирования рельефных поверхностей, например, при отображении облаков, перемещающихся объектов и пр. Наиболее общий способ для реализации этих карт состоит в том, чтобы использовать кубическую (шестисторонную) текстуру.

Прежде чем начать писать соответствующий код, необходимо создать новый проект и выполнить все необходимые приготовления (включая добавление ссылок, переменных устройства, установки окон и пр.). Также понадобятся два объекта (которые включены в DirectX SDK) — модель автомобиля и модель неба skybox2 (с соответствующими текстурами). После чего объявляются следующие переменные:

```
private Mesh skybox = null;
private Material[] skyboxMaterials;
private Texture[] skyboxTextures;
private Mesh car = null;
private Material[] carMaterials;
private Textured carTextures;
private CubeTexture environment = null;
private RenderToEnvironmentMap rte = null;
private const int CubeMapSize = 128;
private readonly Matrix ViewMatrix = Matrix.Translation(0.0f, 0.0f, 13.0f);
```

Итак, мы описали два mesh-объекта для рисования: небо (наша окружающая среда) и автомобиль, на который мы хотим «отражать» окружающую среду. Также необходимо иметь кубическую текстуру, которая сохранит среду, и вспогательный класс для отображения карты среды.

Но, прежде всего, мы должны выяснить, поддерживает ли наша карта кубические текстуры или нет. Поэтому нам нужен новый графический метод инициализации нашего устройства, см. листинг 10.7.

Листинг 10.7. Инициализация устройства для отображения карт

```
public bool InitializeGraphicsO
(
   // Set our presentation parameters
   PresentParameters presentParams = new PresentParametersO;
   presentParams.Windowed = true;
   presentParams.SwapEffeet = SwapEffeet.Discard;
   presentParams.AutoDepthStencilFormat = DepthFormat.D16;
   presentParams.EnableAutoDepthStencil = true;
   // Create our device
   device = new Device(0, DeviceType.Hardware, this,
      CreateFlags.SoftwareVertexProcessing, presentParams);
   device.DeviceReset +=new EventHandler(OnDeviceReset);
   OnDeviceReset(device, null);
   // Do we support cube maps?
   if (!device.DeviceCaps.TextureCaps.SupportsCubeMap)
      return false;
   // Load our meshes
   skybox = LoadMesh(@"..\..\skybox2.x", ref skyboxMaterials,
         ref skyboxTextures);
   car = LoadMesh(@".,\..\car.x", ref carMaterials, ref carlextures)
   return true;
}
```

Главное отличие этого метода заключается в том, что в случае идентификации он возвращает положительное значение логической переменной. По сути, он отслеживает событие сброса устройства и использует новый метод LoadMesh. Прежде чем мы рассмотрим эти методы, необходимо изменить наш метод, чтобы обработать возвращаемое значение:

```
static void Main()
{
   using (Forml frm = new Forml())
   (
      // Show our form and initialize our graphics engine
      frm.Show();
```

```
        if (lfrm.InitializeGraphi.es ())
        {
           MessageBox.Show("Your card does not support cube maps.");
           frm.Closed;
        }
        else
        {
           Application.Run(frm);
        }
    }
}
```

Как видно, здесь нет ничего необычного. Рассмотрим метод, загружающий наш объект (подобно тому, что мы уже делали раньше). После загрузки объекта, материалов и текстур, метод проверяет наличие данных о нормалях (если нет, то добаляет их) и затем возвращает полученный объект. Используйте код, приведенный в листинге 10.8.

Листинг 10.8. Загрузка объекта с данными нормалей.

```
private Mesh LoadMesh(string file,ref Material[] meshMaterials,ref
Texture[] meshTextures)
{
   ExtendedMaterial[] mtrl;
   // Load our mesh
   Mesh mesh = Mesh.FromFile(file, MeshFlags.Managed, device, out mtrl);
   // If we have any materials, store them
   if ((mtrl != null) SS (mtrl.Length > 0))
   {
      meshMaterials = new Material[mtrl.Length];
      meshTextures = new Texture[mtrl.Length];
      // Store each material and texture
      for (int i = 0; i < mtrl.Length; i++)
      {
         meshMaterials[i] = mtrl[i],Material3D;
         if ((mtrl[i].TextureFilename != null) 4&
             (mtrl[i].TextureFilename != string.Empty))
      {
         //We have a texture, try to load it
         meshTextures[i] = TextureLoader.FromFile(device, @"..\..\" +
            mtrl[i].TextureFilename);
      }
   }
}
if ((mesh.VertexFormat & VertexFormats.Normal) != VertexFormats.Normal)
{
```

```
    // We must have normals for our patch meshes
    Mesh tempMesh = mesh.Clone(mesh.Options.Value,
       mesh.VertexFormat | VertexFormats.Normal, device);
    tempMesh.ComputeNormals();
    mesh.Disposed;
       mesh = tempMesh;
    }
    return mesh;
}
```

Теперь, как и в нашем последнем примере, мы должны создать поверхность, на которую будем отображать окружающую среду. Мы будем выполнять это в нашем обработчике события сброса:

```
private void OnDeviceReset(object sender, EventArgs e)
(
    Device dev = (Device)sender;
    rte = new RenderToEnvironmentMapjdev, CubeMapSize, 1,
       Format.X8R8G8B8, true, DepthFormat.D16);
    environment = new CubeTexture(dev, CubeMapSize, 1,
       Usage.RenderTarget, Format.X8R8G8B8, Pool.Default);
}
```

Мы создали вспомогательный класс и кубическую текстуру. Чем больший размер вы пересылаете (в нашем случае константа CubeMapSize), тем более детальной получится карта среды. Во многих случаях более эффективным было бы сохранять карту окружающей среды статически, тогда вы могли бы просто загружать кубическую текстуру из файла (или другого источника данных). Однако, здесь мы хотим показать карту среды, создаваемую динамически, исходя из отображаемой сцены, поэтому мы создаем текстуру для рендеринга.

В последнем примере мы отображали сцену в отдельную текстуру, а в данном случае помимо этого мы по крайней мере шесть раз (для каждой стороны нашей карты куба) осуществим рендеринг нашей сцены,. Добавьте метод, приведенный в листинге 10.9, к вашему приложению.

Листинг 10.9. Рендеринг карты окружающей сцены.

```
private void RenderSceneIntoEnvMap()
{
    // Set the projection matrix for a field of view of 90 degrees
    Matrix matProj;
    matProj = Matrix.PerspectiveFovLH((float)Math.PI * 0.5f, 1.0f,
       0.5f, 1000.0f);
    // Get the current view matrix, to concat it with the cubemap view vectors
```

```
    Matrix matViewDir = ViewMatrix;
    matViewDir.M41 = 0.0f; matViewDir.M42 = 0.0f; matViewDir.M43 = 0.0f;
    // Render the six cube faces into the environment map
    if (environment != null)
       rte.BeginCube(environment);
    for (int i = 0; i < 6; i++)
    {
       rte.Facef(CubeMapFace) i , 1);
       // Set the view transform for this cubemap surface
       Matrix matView = Matrix.Multiply(matViewDir,
          GetCubeMapViewMatrix((CubeMapFace) i));
       // Render the scene (except for the teapot)
       RenderScene(matView, matProj, false);
    }
    rte.End(1);
}
```

Итак, сначала мы создаем матрицу проекции с полем зрения 90 градусов (соответствует углам между гранями куба). Затем получаем матрицу вида, которую мы сохранили для этого приложения и изменяем последнюю строку матрицы таким образом, чтобы можно было объединить ее с матрицами вида для каждой стороны куба.

Далее мы вызываем метод BeginCube, принадлежащий вспомогательному классу, для создания кубической карты окружающей среды. В этом классе имеются и другие методы создания карт среды, включая BeginHemisphere и BeginParabolic (использующие две поверхности, одна для положительной ветви Z, другая для отрицательной), а также BeginSphere (использующий единственную поверхность).

Когда карта среды готова для отображения, мы запускаем цикл для каждой стороны куба. Далее для каждой стороны мы вначале вызываем метод Face, аналогичный методу BeginScene, который мы использовали для нашего устройства, и текстурные рендеры. Метод «отслеживает» событие, когда последняя сторона закончена, а новая готова к отображению. Затем мы объединяем текущую матрицу вида с теми, которые были получены в методе GetCubeMapViewMatrix. Данный метод приведен в листинге 10.10.

Листинг 10.10. Создание матрицы вида, ViewMatrix.

```
private Matrix GetCubeMapViewMatrix(CubeMapFace face)
{
   Vector3 vEyePt = new Vector3(0.0f, 0.0f, 0.0f);
   Vector3 vLookDir = new Vector3();
   Vector3 vUpDir = new Vector3();
   switch (face)
```

```
        case CubeMapFace.PositiveX:
           vLookDir = new Vector3(1.0f, 0.0f, 0.0f);
           vUpDir = new Vector3(0.0f, 1.0f, 0.0f);
           break;
        case CubeMapFace.NegativeX:
           vLookDir = new Vector3(-1.0f, 0.0f, 0.0f);
           vUpDir = new Vector3(0.0f, 1.0f, 0.0f);
           break;
        case CubeMapFace.PositiveY:
           vLookDir = new Vector3(0.0f, 1.0f, 0.0f);
           vUpDir = new Vector3(0.0f, 0.0f,-1.0f);
           break;
        case CubeMapFace.NegativeY:
           vLookDir = new Vector3(0.0f,-1.0f, 0.0f);
           vUpDir = new Vector3(0.0f, 0.0f, 1.0f);
           break;
        case CubeMapFace.PositiveZ:
           vLookDir = new Vector3(0.0f, 0.0f, 1.0f);
           vUpDir = new Vector3(0.0f, 1.0f, 0.0f);
           break;
        case CubeMapFace.NegativeZ:
           vLookDir = new Vector3(0.0f, 0.0f,-1.0f);
           vUpDir = new Vector3(0.0f, 1.0f, 0.0f);
           break;
     }
     // Set the view transform for this cubemap surface
     Matrix matView = Matrix.LookAtLH(vEyePt, vLookDir, vUpDir);
     return matView;
  }
```

Здесь мы просто изменяем параметры вида в зависимости от того, на какую сторону мы смотрим, и возвращаем матрицу вида, составленную на основе этих векторов. После всех этих действий мы отображаем всю сцену без освещенного солнцем автомобиля (параметр «false» в методе RenderScene). Добавьте код, приведенный в листинге 10.11.

Листинг 10.11. Рендеринг всей сцены.

```
private void RenderScene(Matrix View, Matrix Project, bool shouldRenderCar)
{
   // Render the skybox first
   device.Transform.World = Matrix.Scaling(10.0f, 10.0f, 10.0f);
   Matrix matView = View;
   matView.M41 = matView.M42 = matView.M43 = 0.0f;
   device.Transform.View = matView;
   device.Transform.Projection = Project;
```

```
device.TextureState[0].ColorArgumentl  =  TextureArgument.TextureColor;
device,TextureStateiO].ColorOperation  =  TextureOperation.SelectArgl;
device.SamplerState[0].MinFilter  =  TextureFilter.Linear;
device.SamplerState[0].MagFilter  =  TextureFilter.Linear;
device.SamplerState[0].AddressU  =  TextureAddress.Mirror;
device.SamplerState[0].AddressV  =  TextureAddress.Mirror;
// Always pass Z-test, so we can avoid clearing color and depth buffers
device.RenderState.ZBufferFunction  =  Compare.Always;
DrawSkyBoxO;
device.RenderState.ZBufferFunction  =  Compare.LessEqual;
// Render the shiny car
if  (shouldRenderCar)
(
   // Render the car
   device.Transform.View  =  View;
   device.Transform.Projection  =  Project;
   using  (VertexBuffer vb  =  car.VertexBuffer)
   {
      using  (IndexBuffer ib  =  car.IndexBuffer)
      (
         // Set the stream source
         device.SetStreamSource(0, vb, 0,
            VertexInformation.GetFormatSize(car.VertexFormat));
         // And the vertex format and indices
         device.VertexFormat  =  car.VertexFormat;
         device.Indices  =  ib;
         device.SetTexture(0, environment);
         device.SamplerState[0].MinFilter  =  TextureFilter.Linear;
         device.SamplerState[0].MagFilter  =  TextureFilter.Linear;
         device.SamplerState[0].AddressQ  =  TextureAddress.Clamp;
         device.SamplerState[0].AddressV  =  TextureAddress.Clamp;
         device.SamplerState[0].AddressW  =  TextureAddress.Clamp;
         device.TextureState[0].ColorOperation  =
            TextureOperation.SelectArgl;
         device.TextureState[0].ColorArgumentl  =
            TextureArgument.TextureColor;
         device.TextureState[0].TextureCoordinatelndex  =
             (int)TextureCoordinatelndex.CameraSpaceReflectionVector;
         device. TextureState[0].TextureTransform  =
            TextureTransform.Count3;
         device.Transform.World  =  Matrix.RotationYawPitchRoll(
            angle  /  (float)Math.PI, angle  /  (float)Math.PI  *  2.Of,
            angle  /  (float)Math.PI  /  4.0f);
         angle  +=  O.Olf;
         device.DrawIndexedPrimitives(PrimitiveType.TriangleList,
            0, 0, car.NumberVertices, 0, car.NumberFaces);
      }
   }
}
}
```

Первое, что мы здесь делаем, — отображаем объект skybox. Поскольку мы знаем, что размер skybox меньше, чем нам необходимо, мы увеличиваем его в 10 раз. Затем мы задаем текстуру и состояние сэмплера, который необходим для рендеринга текстуры skybox (с этим можно ознакомиться в документах MSDN).

Если вы обратили внимание, мы никогда не очищали устройство в наших примерах. Поскольку нам бы не хотелось вызывать эту процедуру при отображении каждой из сторон куба, мы выполняем ее до отображения карты среды skybox (которая, как мы знаем, будет иметь самое большое значение глубины в нашей сцене) и устанавливаем состояние рендера таким образом, чтобы буфер глубины был всегда доступен во время отображения текстуры skybox. С другой стороны, мы отображаем текстуру skybox и переключаем функцию буфера глубины назад к нормали. Выполнение метода Skybox должно показаться знакомым для нас:

```
private void DrawSkyBoxO
(
    for (int i = 0; i < skyboxMaterials.Length; i++)
    (
        device.Material = skyboxMaterials[i];
        device.SetTexture(0, skyboxTextures[i]);
        skybox.DrawSubset(i) ;
    )
)
```

После рендеринга текстуры skybox необходимо решить, будем ли мы отображать автомобиль, также используя пересылаемый параметр. Если это так, необходимо сбросить процедуру преобразования вида и проекции. Теперь, поскольку мы хотим управлять созданием координат текстур автоматически, мы не будем использовать метод DrawSubset для нашего объекта.

Вместо этого мы будем использовать вершинный и индексный буферы непосредственно, для чего вызовем метод DrawIndexedPrimitives. После этого мы должны быть уверены, что устройство готово к рендерингу объекта.

Вначале мы задаем источник потоковых данных для вершинного буфера из нашего объекта. Также мы, опираясь на наш объект, устанавливаем формат вершин и индексный буфер. Далее мы определяем кубическую текстуру и рисуем наши примитивы. Обратите внимание на следующие строки:

```
device.TextureState[0].TextureCoordinateIndex =
    (int)TextureCoordinateIndex.CameraSpaceReflectionVector;
device.TextureState[0].TextureTransform = TextureTransform.Count3;
```

Так как мы не имеем координат текстуры объекта автомобиль, первая строка подразумевает использование вектора отражения (преобразованного в пространстве камеры) вместо использования координат текстур. Значения вектора автоматически сгенерированы в непрограммируемом конвейере *fixedfunction pipeline,* использующем данные местоположения вершины и нормали к нашей 3D-текстуре.

Теперь мы можем преобразовать координаты и нарисовать наши примитивы. Приложение практически готово к выполнению, но сначала мы должны добавить наш метод рендеринга:

```
protected override void 0nPaint(System.Windows.Forms.PaintEventArgs e)
[
   RenderSceneIntoEnvMap();
   device.BeginSceneO;
   RenderScene(ViewMatrix, Matrix.PerspectiveFovLHf(float)Math.PI / 4,
            this.Width / this.Height, l.Of, 10000.Of), true);
   device. EndSceneO;
   device.Present!);
   this.Invalidate!);
I
```

Для каждого рендера мы сначала отображаем карту окружающей среды и уже затем отображаем нашу «реальную» сцену, включая автомобиль. Приложение готово к запуску. В результате мы должны увидеть на экране освещенный солнцем автомобиль, вращающийся на фоне неба, **рис. 10.4.**

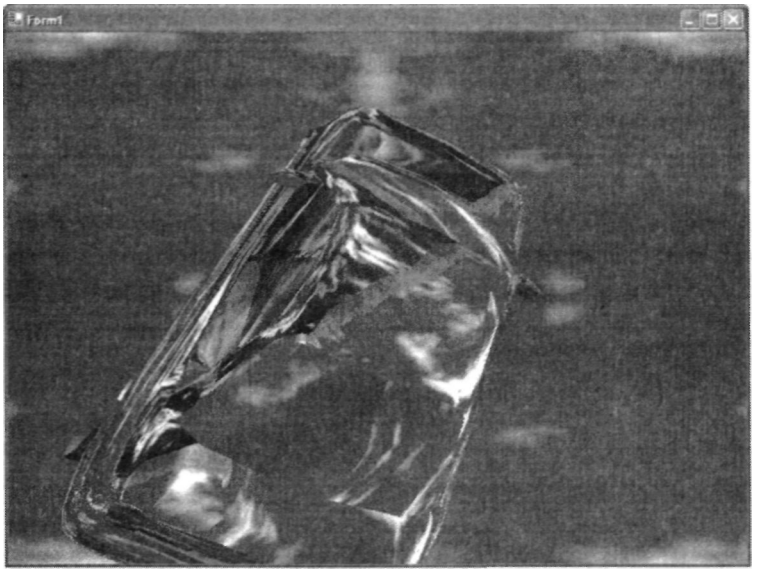


Рис. 10.4. Изображение автомобиля и окружающего фона

Краткие выводы

В этой главе мы изучили следующие аспекты.

- Рисование линий.
- Рисование текста.
- Рендеринг на текстуры.
- Рендеринг карт окружающей среды.

В следующей главе мы рассмотрим более совершенные графические методы и приступим к изучению вершинных и пиксельных шеидеров.

ЧАСТЬ III

БОЛЕЕ СОВЕРШЕННЫЕ МЕТОДЫ ПОСТРОЕНИЯ ГРАФИКИ

Глава 11. Введение в программируемый конвейер, язык шейдеров

Глава 12. Использование языка шейдеров HLSL

Глава 13. Рендеринг скелетной анимации

Глава 1 1 . Введение в программируемый конвейер, язык шеидеров

До этого момента для процедуры рендеринга мы использовали непрограммируемый конвейер. До создания DirectX 8.0 это было единственным способом отображать что-либо на экране. Непрограммируемый конвейер является по существу набором правил и параметров, которые управляют представленными типами данных. Тем не менее, его использование не позволяет реализовать многие возможности процедур и опций рендеринга, например, при использовании освещения.

С появлением DirectX 8.0 появилась возможность работать с программируемым конвейером, которая позволила разработчикам управлять параметрами конвейера. Все это привело к появлению шеидеров или языка построения теней в компьютерной графике.

Теперь разработчики могли управлять обработкой вершин с помощью *вершинных шеидеров* и обработкой пикселей с помощью *пиксельных шеидеров*. Программы, использующие язык шеидеров оказались весьма мощными, но на тот момент не совсем удобными. Синтаксис языка напоминал ассемблирование и требовал некоторых доработок и упрощений.

В Управляемом DirectX 9 был реализован язык HLSL — язык шеидеров высокого уровня. Язык HLSL был более простым для разработчиков и напоминал язык C, который мог быть откомпилирован в реальный код шейдера. Таким образом, могли быть реализованы все преимущества программируемого конвейера в более доступной и удобной форме. В этой главе мы рассмотрим основные особенности языка шеидеров HLSL, включая.

Использование программируемого конвейера.

Преобразование вершин с помощью шеидеров.

- Использование пиксельных шеидеров.

Рендеринг треугольника с использованием программируемого конвейера

В прошлых главах мы неоднократно упоминали об использовании непрограммируемого конвейера, но подробно не описывали его. Непрограммируемый конвейер управляет тем, как вершины отображаются, как они преобразованы и освещены и т. д. Когда мы определяем свойства вершинного формата в устройстве, мы сообщаем устройству, что хотим отобразить вершину тем или иным способом, основываясь на данных формата вершины.

Одним из главных недостатков такого подхода являлось то, что для каждого отдельного свойства или возможности, которую графическая карта должна реализовать, необходимо было использовать непрограммируемый API интерфейс. С быстрым развитием аппаратной части компьютерной графики (даже более быстрым, чем, например, процессоров), число интерфейсов API быстро бы вышло из-под контроля. Кроме того, не было возможности управлять интерфейсами, что весьма необходимо для разработчиков компьютерной графики (особенно разработчиков игр).

Если вспомнить, одним из первых приложений, рассмотренных в этой книге, было отображение вращающегося треугольника. Это было весьма просто сделать с использованием непрограммируемого конвейера. Теперь для этого примера мы будем использовать программируемый конвейер. Как обычно, вначале создаем новые формы windows-приложения, объявляем переменную устройства, стиль окна и т. д. Для этого добавим следующие переменные:

```
private VertexBuffer vb = null;
private Effect effect = null;
private VertexDeclaration decl = null;
// Our matrices
private Matrix worldMatrix;
private Matrix viewMatrix;
private Matrix projMatrix;
private float angle = 0.0f;
```

Естественно, мы будем записывать данные вершин в вершинном буфере. Необходимо также сохранить матрицы преобразования до того момента, когда мы будем использовать их в программируемом конвейере. Вторая и третья переменные, описанные здесь, — новые для нас. Effect object— главный объект, который вы будете использовать при программировании на языке шейдеров HLSL. Класс объявления вершин VertexDeclaration подобен вершинному формату в непрограммируемом конвейере. Данный класс сообщает приложению Direct3D runtime о размере и типах данных, которые будут считываться из вершинного буфера.

Поскольку это приложение будет использовать относительно новую возможность графических карт, а именно программируемый конвейер, вполне возможно, что ваша графическая плата не будет поддерживать эту опцию. Если это так, необходимо подключить ссылки на эмулированное устройство, которое включено в DirectX SDK, другим словами, создать программную эмуляцию устройства. Для данного случая работа API будет выполняться несколько медленнее, и возникает необходимость в использовании более быстрой процедуры инициализации нашего устройства, см. листинг 11.1.

Листинг 11.1. Инициализация графики, проверка поддерживаемых режимов (Fallback).

```
public bool InitializeGraphics()
{
   // Set our presentation parameters
   PresentParameters presentParams = new PresentParameters();
   presentParams.Windowed = true;
   presentParams.SwapEffeet = SwapEffeet.Discard;
   presentParams.AutoDepthStencilFormat = DepthFormat.D16;
   presentParams.EnableAutoDepthStencil = true;
   bool canDoShaders = true;
   // Does a hardware device support shaders?
   Caps hardware = Manager.GetDeviceCapsf0, DeviceType.Hardware);
   if (hardware.VertexShaderVersion >= new Version(1, 1))
   {
      // Default to software processing
      CreateFlags flags = CreateFlags.SoftwareVertexProcessing;
      // Use hardware if it's available
      if (hardware.DeviceCaps.SupportsHardwareTransformAndLight)
         flags = CreateFlags.HardwareVertexProcessing;
      // Use pure if it's available
      if (hardware.DeviceCaps.SupportsPureDevice)
         flags |= CreateFlags.PureDevice;
      // Yes, Create our device
      device = new Device(0, DeviceType.Hardware, this, flags, presentParams';
      }
      else
      {
         //No shader support
         canDoShaders = false;
         // Create a reference device
         device = new Device(0, DeviceType.Reference, this,
         CreateFlags.SoftwareVertexProcessing, presentParams);
      }
      // Create our vertex data
      vb = new VertexBuffer(typeof(CustomVertex.PositionOnly), 3, device,
         Usage.Dynamic | Usage.WriteOnly, CustomVertex.PositionOnly.Format,
         Pool.Default);
      vb.Created += new EventHandler(this.OnVertexBufferCreate);
      OnVertexBufferCreate(vb, null);
      // Store our project and view matrices
      projMatrix = Matrix.PerspectiveFovLH((float)Math.PI / 4,
         this.Width / this.Height, 1.0f, 100.0f);
      viewMatrix = Matrix.LookAtLH(new Vector3(0,0, 5.0f), new Vector3(),
         new Vector3(0,1,0));
      // Create our vertex declaration
```

```
        VertexElement[] elements = new VertexElement[]
            {
                new VertexElement(0, 0, DeclarationType.Float3,
                DeclarationMethod.Default,
                DeclarationUsage.Position, 0),
                VertexElement.VertexDeclarationEnd
            };
    decl = new VertexDeclaration(device, elements);
    return canDoShaders;
}
```

Данный код предполагает использование для рендеринга сцены адаптера, выбранного по умолчанию (нулевое значение параметра). При этом, здесь опущена процедура перебора, вместо этого, как уже сказано, выбран адаптер, используемый по умолчанию. Вы можете при необходимости изменить код в соответствии с используемой графической картой. Перед созданием устройства, мы, как и раньше, проверяем поддерживаемые им возможности, если вспомнить, данная структура называлась Caps (возможности).

В настоящем приложении, поскольку здесь мы используем для рендеринга вершин программируемый конвейер, необходимо убедиться, что наша карта поддерживает по крайней мере шейдеры первого поколения. С появлением новых версий API версии вершинных и пиксельных шейдеров также обновлялись. Например, DirectX 9 позволяет нам использовать шейдеры версии 3.0 и старше (хотя в настоящее время нет подходящих графических карт, поддерживающих, к примеру, третью версию). Первое поколение шейдеров имело версию 1.0. В DirectX 9 эта устаревшая версия была заменена версией 1.1, и это то, с чем мы здесь будем работать.

Предполагая, что карта поддерживает эту версию, мы можем создать «более продвинутый» тип устройства. Мы задаем по умолчанию программную обработку вершин, а если устройство поддерживает и аппаратную обработку, будем использовать ее. В случае, если плата не поддерживает программы шейдеров, нам ничего не остается, кроме как использовать программное устройство — эмулятор, которое поддерживает режим runtime, хотя это будет намного медленнее.

Далее необходимо создать вершинный буфер для нашего отображаемого треугольника. Обработчик событий в этом случае будет иметь следующий вид:

```
private void OnVertexBufferCreate(object sender, EventArgs e)
{
    VertexBuffer buffer = (VertexBuffer)sender;
    CustomVertex.PositionOnly[] verts = new CustomVertex.PositionOnly[3];
```

```
    verts[0].SetPosition(new Vector3(0.0f, 1.0f, 1.0f));
    verts[1].SetPosition(new Vector3(-1.0f, -1.0f, 1.0f));
    verts[2].SetPosition(new Vector3(1.0f, -1.0f, 1.0f));
    buffer.SetData(verts, 0, LockFlags.None);
}
```

После создания вершинного буфера необходимо сохранить результаты преобразования координат для дальнейшего использования. Данные преобразования выполняются тем же самым способом, что и раньше, с единственным отличием — теперь для записи этих данных мы не будем использовать свойство преобразования устройства.

Итак, мы дошли до процедуры объявления вершин, «сообщающей» Direct3D всю необходимую информацию о вершинах, которая будет пересылаться в программируемый конвейер. При создании объекта класса VertexDeclaration в используемое устройство пересылается массив вершинных элементов, каждый из которых описывает один компонент данных вершин. Рассмотрим следующий конструктор для элемента вершины:

```
public VertexElement ( System.Int16 stream , System.Int16 offset ,
   Microsoft.DirectX.Direct3D.DeclarationType declType ,
   Microsoft.DirectX.Direct3D.DeclarationMethod declMetftod ,
   Microsoft.DirectX.Direct3D.DeclarationUsage declUsage ,
   System.Byte usagelndex )
```

Первый параметр *stream* — поток данных о вершинах. Когда мы вызывали метод SetStreamSource, этот параметр представлял собой поток, пересылаемый в вершинный буфер. До сих пор мы хранили все данные в одном вершинном буфере, используя один поток; теперь мы можем проводить рендеринг объекта, используя данные из различных вершинных буферов с различными потоками данных. В данном примере, поскольку мы имеем только один вершинный буфер (поток с нулевым порядковым номером), мы задаем нулевое значение для этого параметра.

Второй параметр *offset* — смещение в буфере, где хранятся данные. Поскольку в нашем случае имеется только один тип данных вершинного буфера, данное значение также будет нулевым. При использовании нескольких типов данных значение смещения задается для каждого. Например, если первый компонент — местоположение вершины (три значения с плавающей точкой), а второй компонент — нормаль (также три значения с плавающей точкой), тогда первый компонент будет иметь смещение «О» (поскольку это первый компонент в буфере), тогда как нормаль — смещение «12» (четыре байта на каждую цифру).

Третий параметр *declType* сообщает приложению Direct3D тип используемых данных. Так как в этом примере мы используем только местоположение вершины, можно использовать тип Float3 (этот тип мы обсудим позже).

Четвертый параметр *declMethod* описывает используемый метод. В большинстве случаев (если не используются примитивы более высокого порядка), метод устанавливается по умолчанию.

Пятый параметр *declUsage* описывает использование компонентов, таких как: местоположение, нормаль, цвета и пр. Для описания местоположения используются три числа с плавающей запятой.

Последний параметр изменяет данные использования, что позволяет определять множественные типы использования. В большинстве случаев это нулевое значение.

Важно обратить внимание, что ваш массив элементов вершины должен иметь в качестве последнего значения запись VertexElement.VertexDeclarationEnd. Так, для простого объявления вершины можно использовать поток с нулевым значением номера, состоящий из трех чисел с плавающей запятой, представляющих месторасположения вершины. После создания массива элементов вершин можно создавать объект объявления вершины. Окончательно, программа возвращает булеву переменную, имеющую значение «true», если аппаратное устройство поддерживает шейдеры, и значение «false», если используется эмулированное устройство.

Таким образом, необходимо вызвать метод инициализации, использующий эту булеву переменную. Перепишите метод следующим образом:

```
static void Main()
{
   using (Forml frm = new Forml())
   {
      // Show our form and initialize our graphics engine
      frm.Show();
      if  (Ifrra.InitializeGraphics())
      {
         MessageBox.Show("Your card does not support shaders. " +
            "This application will run in ref mode instead.");
      }
      Application.Run(frm);
   }
}
```

Теперь осталось вызвать метод рендеринга сцены, который мы использовали неоднократно, для этого необходимо переписать метод OnPaint следующим образом:

```
protected override void OnPaint(System.Windows.Forms.PaintEventArgs e)
{
   device.Clear(ClearFlags.Target | ClearFlags.ZBuffer, Color.CornflowerBlue,
   1.Of, 0);
```

```
    UpdateWorldO;
    device.BeginScene ();
    device.SetStreamSource(0, vb, 0);
    device.VertexDeclaration = decl;
    device.DrawPrimitives(PrimitiveType.TriangleList, 0, 1);
    device.EndScenef);
    device.Present ();
    this.Invalidate));
}
```

Затем, после очистки устройства, мы должны вызвать метод преобразования мировых координат, который в данном примере определяет приращение угловой переменной и изменяет сохраненную матрицу преобразований:

```
private void UpdateWorld()
{
   worldMatrix = Matrix.RotationAxis(new Vector3(angle / ((float)Math.PI *
   2.0f),
      angle / ((float)Math.PI * 4.0f), angle / ((float)Math.PI * 6.0f)),
      angle / (float)Math.PI);
   angle += 0.1f;
)
```

Достаточно знакомый код, за исключением того, что мы устанавливаем свойство объявления вершины vertex declaration вместо свойства формата вершины vertex format. Выполняя приложение в том виде, каком оно есть на данный момент, мы ничего не увидим кроме синего экрана. С чем это может быть связано? Все достаточно просто, приложение Direct3D runtime понятия не имеет, что вы хотите сделать. Мы должны написать реальную «программу» для программируемого конвейера.

Итак, добавьте новый пустой текстовый файл simple.fx к вашему проекту. В этот файл мы будем сохранять программу, написанную на языке HLSL. Теперь необходимо добавить следующий HLSL код в этом файле:

```
// Shader output, position and diffuse color
struct VS_0UTPUT
{
   float4 pos : POSITION;
   float4 diff : COLOR0;
};
// The world view and projection matrices
float4x4 HorldViewProj : WORLDVIEWPROJECTION;
float Time = 1.0f;
// Transform our coordinates into world space
VS OUTPUT Transform)
```

```
float4 Pos : POSITION)
{
    // Declare our return variable
    VS_0UTPUT Out = (VS_OUTPUT)0;
    // Transform our position
    Out.pos = mul(Pos, WorldViewProj);
    // Set our color
    Out.diff.r = 1 - Time;
    Out.diff.b = Time * WorldViewProj[2].yz;
    Out.diff.ga = Time * WorldViewProj[0].xy;
    // Return
    return Out;
I
```

Легко заметить, что код HLSL напоминает язык C (и C#). Вначале мы объявляем структуру, которая будет связывать выходные данные вершин. Процедура объявления переменных несколько отличается от привычного нам алгоритма. К каждой переменной была добавлена семантика (другими словами, тэги, указывающие на переменную класса usage: в нашем случае это местоположение и первоначальный цвет, соответственно position и color).

Возникает вопрос, почему в структуре вывода находятся и местоположение, и цвет, тогда как вершинный буфер содержит данные только о местоположении? Поскольку структура вывода содержит и местоположение, и цвет (вместе с соответствующей семантикой), Direct3D будет «знать» что рендерингу подлежат новые данные, возвращенные из программы обработки вершин или вершинных шейдеров, вместо данных, сохраненных в вершинном буфере.

Далее, имеются две «глобальные» переменные, затрагивающие различные преобразования: первая — комбинацию матрицы вида, проекции и пр., и вторая — время анимирования цвета треугольника. Во время выполнения приложения необходимо обновлять или перезаписывать все эти параметры для каждого кадра.

Итак, у нас имеется реальный код вершинного шейдера. Как можно видеть, он возвращает структуру, которую мы уже создали, и принимает в качестве параметра положение вершин. Этот метод будет вызываться для каждой из вершин, находящихся в вершинном буфере (в нашем случае трех). Обратите внимание, что входная переменная также имеет описание, указывающее приложению Direct3D тип используемых данных.

Сам по себе код внутри приведенного метода весьма прост. Мы объявляем возвращаемую переменную Out. Далее выполняется процедура преобразования, при котором начальное значение местоположения умножается на сохраненную матрицу преобразования. Здесь используется встроенная функция mull, причем ранг матрицы преобразования — float4x4, а вектора местоположения — float4. Необходимо строго согласовывать форматы при перемножении в соответствии с известными операциями

над матрицами, в противном случае, оператор не выполнит данную операцию из-за несоответствия типов.

После этого мы определяем цвета, используя простую формулу для каждого компонента цвета. Сначала устанавливается красный, затем синий и в конце зеленый. Таким образом, установив значения расположения и цвета, мы имеем заполненную структуру, которая будет использоваться в дальнейших операциях.

ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ В ЯЗЫКЕ HLSL И ВСТРОЕННЫЕ ТИПЫ

Обратите внимание на то, что мы используем некоторые встроенные типы, которых нет в языках C или C#, а именно float4 и float4x4. Следует отметить, что язык шейдеров поддерживает следующие скалярные типы.

Bool — булева переменная, «true» или «false».
Int — 32-разрядное целое число со знаком.
Half— 16-разрядное число с плавающей запятой.
Float — 32-разрядное число с плавающей запятой.
Double — 64-разрядное число с плавающей запятой.

Объявление переменных этих типов просходит также, как и в языке C#. Однако, добавление отдельного целого числа в конце одного из этих скалярных типов объявит «векторный» тип. Векторные типы могут использоваться как векторы или массивы. Например:

```
float4  pos;
```

Данная строка объявляет переменную Vector4, которая содержит информацию о месторасположении. Если мы хотим объявить двухмерный вектор, можно записать следующее:

```
float2  someVector;
```

Вы можете обращаться к значениям этой переменной как к массиву:

```
pos[0]  =  2.Of;
```

Также можно обращаться к этому типу аналогичным образом, что и к вектору в C#:

```
pos.x  =  2.Of;
```

В этом случае мы можем обращаться к одному или к нескольким компонентам вектора. Это называется «swizzling» или обращение по адресам. Мы можем использовать компоненты (x,y,z,w) вектора, также

как составляющие цвета (r,g,b,a); однако, мы не можем смешивать эти компоненты в данном процессе. Пример корректной записи:

```
pos.xz = O.Of;
pos.rg += pos.xz;
```

Пример некорректной записи:

```
pos.xg = O.Of;
```

Запись xg недействительна, поскольку смешиваются компоненты разных типов.

Переменные в языке шейдеров могут иметь модификаторы похожие на модификаторы языка C или C#. Вы можете объявлять константы тем же самым образом, что и ранее:

```
const float someConstant = 3. Of;
```

Вы можете совместно использовать переменные в различных программах:

```
shared float someVariable = 1.Of;
```

При необходимости более подробую информацию относительно HLSL можно найти в документации DirectX SDK.

Использование шейдеров для рендеринга, использование техник «TECHNIQUE»

Теперь, когда мы написали программу вершинного шейдера, необходимо определить «точки входа» в данную программу и написать соответствующий алгоритм. Для этого мы можем использовать процедуру, называемую «техника» («technique»). Под техникой понимается способ (или последовательность кода), реализующий ту или иную функцию шейдера. Такой прием позволяет осуществить один или несколько проходов, каждый раз с помощью кода HLSL определяя состояние устройства, а также вершинные и пиксельные шейдеры. Рассмотрим простой алгоритм, который применим к нашему приложению. Добавьте следующий код к вашему файлу simple.fx:

```
technique TransformDiffuse
{
   pass PO
    {
```

S Зак. 604

```
CullMode = None;
// shaders
VertexShader = compile vs_1_1 Transform();
PixelShader = NULL;
```

Здесь мы объявляем технику TransformDifruse, позволяющую преобразовать вершину и добавить диффузный цвет. В данной технике определен только один цикл. Режим отбрасывания невидимых сторон Cull не включен, так что мы можем видеть обе стороны нашего единственного треугольника. Если вспомнить, мы уже писали аналогичный код на языке C#.

Теперь мы попробуем «запрограммировать» обработку вершин, используя вершинный шейдер версии 1.1 (в программе vs_1_1). При использовании версии 2.0 в программе нужно указать vs_2_0. Пиксельный шейдер определяется как ps_2_0.

Так как мы не используем здесь обработку пиксела, мы определяем: PixelShader = NULL. Далее мы должны переписать C# код, чтобы использовать эту технику и соответствующий цикл.

Для начала используем объект Effect HLSL-кода, объявленный ранее как основной. Добавьте следующие строки к методу InitializeGraphics (после создания устройства):

```
// Create our effect
effect = Effect.FromFile(device, @". .\..\simple.fx", null,
ShaderFlags.None,
   null);
effect.Technique = "TransformDiffuse";
```

Объект Effect создается из файла simple.fx. Затем мы объявляем технику Effect для ее использования в нашей HLSL-программе. Если вы помните, у нас имелись две переменные, которые необходимо было перезаписывать каждый раз при смене кадра. Поэтому добавим следующий код в конце метода UpdateWorld:

```
Matrix worldViewProj = worldMatrix * viewMatrix * projMatrix;
effect.SetValue("Time", (float)Math.Sin(angle / 5.0f));
effect.SetValue("WorldViewProj", worldViewProj);
```

Здесь мы объединяем наши сохраненные матрицы и изменяем переменные в коде HLSL. Этот метод вызывается каждый раз при смене кадра, поэтому переменные будут обновляться автоматически. Теперь осталось переписать процедуру рендеринга, используя программу вершинного шейдера. Перепишите ваш метод DrawPrimitives следующим образом:

```
int numPasses = effect.Begin(O);
for (int i = 0; i < numPasses; i++)
{
    effect.Pass(i);
    device.DrawPrimitives(PrimitiveType.TriangleList, 0, 1);
}
effect.End();
```

Вначале мы вызываем метод effectBegin для нашей техники Effect. Используя флажки для этого метода, мы могли бы выбрать режим, позволяющий не сохранять отдельные состояния, но для этого примера это не так важно.

Метод возвращает число циклов в используемой технике и создает соответствующий цикл рендеринга (в нашем случае один). Перед тем как создать рисунок, необходимо вызвать метод прохода или цикла Pass для нашего объекта. Единственный принимаемый в этом методе параметр — индекс *i*. Данный метод подготавливает устройство к процедуре рендеринга для обозначенного цикла, обновляет состояние устройства и устанавливает методы для вершинных и пиксельных шейдеров. И, наконец, знакомый нам метод DrawPrimitives рисует примитивы (в нашем случае треугольник).

В конце метода необходимо добавить команду завершения efFectEnd. При запуске приложения на экране отобразится вращающийся треугольник с последовательно изменяющимися цветами, см. **рис. 11.1.**

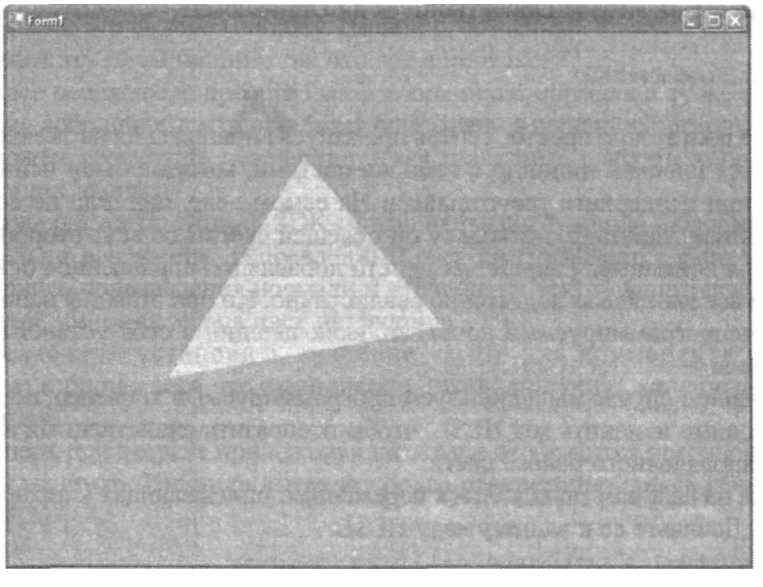


Рис. 11.1. Красочный вращающийся треугольник

Использование программируемого конвейера для рендеринга mesh-объектов

Пример с отображением треугольника является чересчур простым. Попробуем отобразить объект mesh, используя программируемый конвейер.

Для этого заменим переменные в разделе объявления вершин и вершинном буфере на следующее:

```
private Mesh mesh = null;
```

Одно из преимуществ mesh-объектов состоит в том, что при рендеринге они автоматически задают параметр объявления вершин vertex declaration. Поэтому в нашем случае необходимо убрать в используемом коде все ссылки на вершинные буферы или параметры объявления вершин.

Необходимо также изменить метод инициализации для создания mesh-объекта, для этого после создания устройства и объекта добавьте следующий код:

```
// Create our cylinder
mesh = Mesh.Cylinder(device, 1.5f, 1.5f, 1.5f, 36, 36);
```

И в завершении необходимо изменить метод рисования объекта. Замените вызов метода DrawPrimitives на DrawSubset:

```
mesh.DrawSubset(0);
```

Все достаточно просто. Теперь при запуске приложения на экране отобразится плоский цилиндр с теми же цветами, которые были использованы при рендеринге треугольника. На самом деле, цилиндр не совсем напоминает цилиндр, поскольку он освещен светом со всех сторон и не кажется объемным. Раньше мы просто добавляли направленное освещение, и все выглядело достаточно реалистично. Но при этом мы использовали непрограммируемый конвейер, включающий в себя установку освещения.

В нашем случае мы используем программируемый конвейер, поэтому необходимо изменить код HLSL, чтобы отобразить единственный источник направленного белого света.

Для начала нам понадобится переменная, описывающая направление света. Добавьте ее к вашему коду HLSL:

```
// The direction of the light in world space
float3 LightDir = (O.Of, O.Of, -l.Of);
```

Теперь мы можем заменить существующий метод на следующий:

```
// Transform our coordinates into world space
VSJIUTPUT Transform)
   float4 Pos : POSITION,
   float3 Normal : NORMAL )
{
   // Declare our return variable
   VSJXJTPUT Out = (VSJXJTPUT) 0;
   // Transform the normal into the same coord system
   float4 transformedNormal = mul(Normal, WorldViewProj);
   // Set our color
   Out.diff.rgba = l.Of;
   Out.diff *= dot(transformedNormal, LightDir);
   // Transform our position
   Out.pos = mul(Pos, WorldViewProj);
   // Return
   return Out;
}
```

Обратите внимание, что в метод включен новый параметр, а именно данные о нормали («normal»). Необходимо убедиться, что каждая вершина, которую мы прогоняем в процессе обработки, имеет данные о нормали, иначе это вызовет ошибку. Цилиндр, созданный классом mesh, уже содержит эту информацию, так что все в порядке.

Далее необходимо преобразовать координаты нормали в ту же самую систему координат, в которой будет описываться вершина, поскольку мы не сможем рассчитать освещение, имея различные системы координат для наших данных.

В литературе по DirectX SDK можно найти следующую информацию, касательно освещения: направленный свет рассчитывается с помощью скалярного произведения нормали к вершине на вектор направления освещения, умножая это на цвет индикатора.

Вы можете установить значение «l.Of» для компонента цвета (r,g,b,a) и произвести это вычисление, чтобы получить конечный цвет цилиндра.

Теперь при запуске приложения на экране отобразится вращающийся белый цилиндр. Цилиндр выглядит более реалистично чем раньше, см. **рис.11.2.**

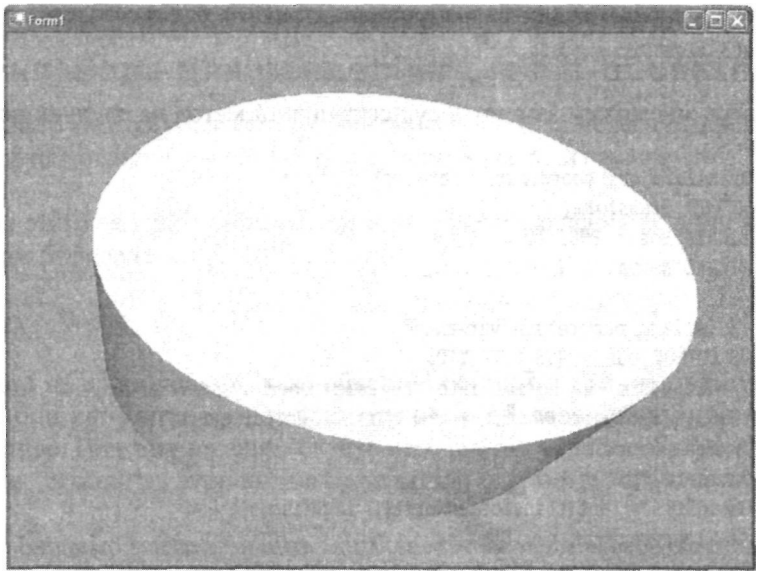


Рис. 11.2. Вращающийся цилиндр, выполненный на языке шейдеров

Изменяя цвет освещения можно получать различные цветовые эффекты.

Использолвание языка HLSL для создания пиксельного шейдера

Вершинные шейдеры — это только часть того, что можно реализовать, используя программируемый конвейер. Было бы весьма заманчиво рассмотреть цвета каждого пиксела. Для этого мы возьмем пример MeshFile, рассмотренный нами в главе 5 (Рендеринг mesh-объектов) и изменим его с помощью программируемого конвейера. В примере из гл. 5 помимо обработки вершин мы использовали обработку данных текстур, поэтому попытка применить пиксельный шейдер может оказаться весьма наглядной.

После загрузки данного проекта нам нужно будет внести несколько изменений в исходник программы, чтобы можно было использовать программируемый конвейер.

Вначале объявляем переменные для объекта Effect и матриц преобразования:

```
private Effect effect = null;
// Matrices
private Matrix worldMatrix;
private Matrix viewMatrix;
private Matrix projMatrix;
```

Необходимо также переписать метод инициализации эмулируемого устройства в случае, если графическая карта не поддерживает шейдеры. Метод инициализации приведен в листинге 11.2.

Листинг 11.2. Инициализация графики, проверка поддерживаемых режимов (Fallback).

```
public bool InitializeGraphics ()
(
   // Set our presentation parameters
   PresentParameters presentParams = new PresentParameters();
   presentParams.Windowed = true;
   presentParams.SwapEffeet = SwapEffect.Discard;
   presentParams.AutoDepthStencilFormat = DepthFormat.D16;
   presentParams.EnableAutoDepthStencil = true;
   bool canDoShaders = true;
   // Does a hardware device support shaders?
   Caps hardware = Manager.GetDeviceCaps(3, DeviceType.Hardware);
   if ((hardware.VertexShaderVersion >= new VersionU, 1)) &&
      (hardware.PixelShaderVersion >= new Version(1, 1)))
   {
      // Default to software processing
      CreateFlags flags = CreateFlags.SoftwareVertexProcessing;
      // Use hardware if it's available
      if (hardware.DeviceCaps.SupportsHardwareTransformAndLight)
      flags = CreateFlags.HardwareVertexProcessing;
      // Use pure if it's available
      if (hardware.DeviceCaps.SupportsPureDevice)
         flags |= CreateFlags.PureDevice;
      // Yes, Create our device
      device = new Device(0, DeviceType.Hardware, this, flags, presentParams)
   }
   else
   (
      // No shader support
      canDoShaders = false;
      // Create a reference device
      device = new Device(0, DeviceType.Reference, this,
      CreateFlags.SoftwareVertexProcessing, presentParams);
   I
   // Create our effect
   effect = Effect.FromFile(device, @"..\..\simple.fx", null,
   ShaderFlags.None, null);
   effect.Technique = "TransformTexture";
   // Store our project and view matrices
   projMatrix = Matrix.PerspectiveFovLH((float)Math.PI / 4,
```

```
        this.Width / this.Height, 0f, 10000.0f);
    viewMatrix = MatrixLookAtLH(new Vector3(0,0, 580.0f), new Vector3(),
        new Vector3(0,1,0));
    // Load our mesh
    LoadMesh(@"..\..\tiny.x");
    return canDoShaders;
}
```

Данный метод аналогичен методу инициализации, который мы уже использовали в этой главе, только помимо вершинных шейдеров здесь проверяется возможность поддержки и пиксельных шейдеров (как и раньше, возвращается булева переменная). Для изменения основного метода Main мы можем обратиться к тому же алгоритму, который использовали в этой главе при отображении треугольника.

Необходимо отметить, что, поскольку метод SetupCamera используется только в непрограммируемом конвейере, мы можем исключить его из программы при преобразовании координат или установке освещения. Естественно при этом необходимо также удалить соответствующий вызов в методе OnPaint.

И последнее, мы должны заменить вызов рисунка DrawMesh соответствующим кодом рендеринга, написанном на языке HLSL и приведенном в листинге 11.3.

Листинг 11.3. Код вызова рисунка mesh-объекта, написанный на языке HLSL.

```
private void DrawMesh(float yaw, float pitch, float roll, float x, float y.
float z)
{
   angle += 0.01f;
   worldMatrix = Matrix.RotationYawPitchRoll(yaw, pitch, roll)
      * Matrix.Translation(x, y, z);
   Matrix worldViewProj = worldMatrix * viewMatrix * projMatrix;
   effect.SetValue("WorldViewProj", worldViewProj);
   int numPasses = effect.Begin(0);
   for (int iPass = 0; iPass < numPasses; iPass++)
   {
      effect.Pass(iPass);
      for (int i = 0; i < meshMaterials.Length; i++)
      {
         device.SetTexture(0, meshTextures[i]);
         mesh.DrawSubset (i);
         }
         }
   effect.End();
}
```

Данный метод вызывается при каждой смене кадра. Как видно из метода, мы объединяем матрицы преобразования и изменяем в соответствии с этим код HLSL.

Затем мы отображаем каждое подмножество объекта mesh для каждого цикла техники. Однако, мы до сих пор не объявили и не создали для этого приложения исходник на языке HLSL, так что необходимо добавить новый пустой файл «simple.fx» и добавить код, приведенный в листинге 11.4.

Листинг 11.4. Код HLSL для рендеринга текстурных объектов.

```
// The world view and projection matrices
float4x4 WorldViewProj : WORLDVIEWPROJECTION;
sampler TextureSampler;

// Transform our coordinates into world space
void Transform(
   in float4 inputPosition : POSITION,
   in float2 inputlexCoord : IEXCOORD0,
   out float4 outputPosition : POSITION,
   out float2 outputTexCoord : TEXCOORDO
   )
{
   // Transform our position
   outputPosition = mul(inputPosition, WorldViewProj);
   // Set our texture coordinates
   outputTexCoord = inputlexCoord;
}
void TextureColor(
in float2 textureCoords : TEXCOORDO,
out float4 diffuseColor : COLORO)
{

   // Get the texture color
   diffuseColor = tex2D(TextureSampler, textureCoords);
};
technique TransformTexture
{
   pass PO
   {
      // shaders
      VertexShader = compile vs_1_1 Transform();
      PixelShader = compile ps_1_1 TextureColor();
   }
}
```

Легко заметить, чем отличаются программы шейдеров. Вместо сохранения возвращаемых значений в структуре, мы просто добавили выходные параметры в раздел объявления метода.

Для программирования вершины нас интересуют только данные ее положения и координаты текстуры.

Используемый нами пиксельный шейдер принимает координаты текстуры для соответствующего пиксела и возвращает цвет данного отображаемого пиксела.

Для нашего случая мы используем заданный по умолчанию цвет текстуры, и встроенный метод tex2D производит выборку текстуры в данном наборе координат и возвращает цвет в этой точке.

Результат использования пиксельного шейдера для нашего mesh-объекта приведен на **рис.** 11.3.

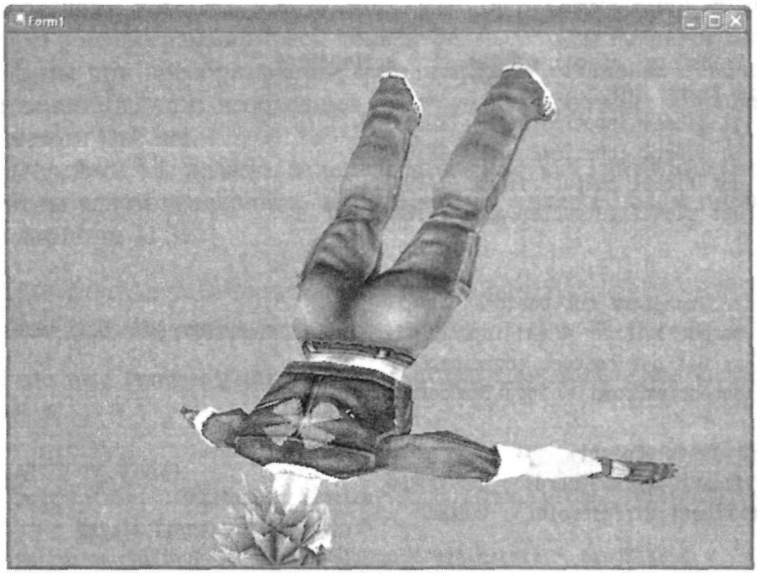


Рис. 11.3. Результат рендеринга mesh-объекта с использованием пиксельного шейдера

На самом деле, глядя на рисунок, мы не видим пока каких-либо существенных изменений по сравнению с картинками, построенными на базе непрограммируемого конвейера. Попробуем использовать еще одну технику для HLSL-программы. Добавьте следующий метод к вашему коду HLSL:

```
void InverseTextureColor(
in float2 textureCoords : TEXCOORDO,
```

```
out float4 diffuseColor : COLORO)
{
   // Get the inverse texture color
   diffuseColor = 1.Of — tex2D(TextureSampler, textureCoords);
};
technique TransformInverseTexture
{
   pass PO
   {
      // shaders
      VertexShader = compile vs_l_l Transform();
      PixelShader = compile ps_l_l InverselextureColor();
   }
}
```

Данный код аналогичен нашему первому пиксельному шейдеру с той лишь разницей, что теперь при определении цвета мы вычитаем выборочный цвет из значения «1.Of». Поскольку значение «lOf» рассматривается как «fully on» для цвета, произведя вычитание, мы обращаем цвет пиксела. Помимо этого мы будем использовать еще одну технику TransformlnverseTexture, которая отличается только методом вызываемого шейдера.

Теперь нам необходимо переписать наш основной C# код, чтобы иметь возможность переключать эти методы. Для этого необходимо написать обработку нажатия клавиш «1» и «2» на клавиатуре:

```
protected override void OnKeyPress(KeyPressEventArgs e)
{
   switch (e.KeyChar)
   {
      case '1':
         effect.Technique = "TransformTexture";
         break;
      case '2':
         effect.Technique = "TransformlnverseTexture";
         break;
   }
   base.OnKeyPress (e);
}
```

Теперь запустите приложение и нажмите клавишу «2». Объект теперь выглядит как негатив, **рис.11.4.**

Пример программы, позволяющей изменять цвета объекта, включен в прилагаемый CD диск.

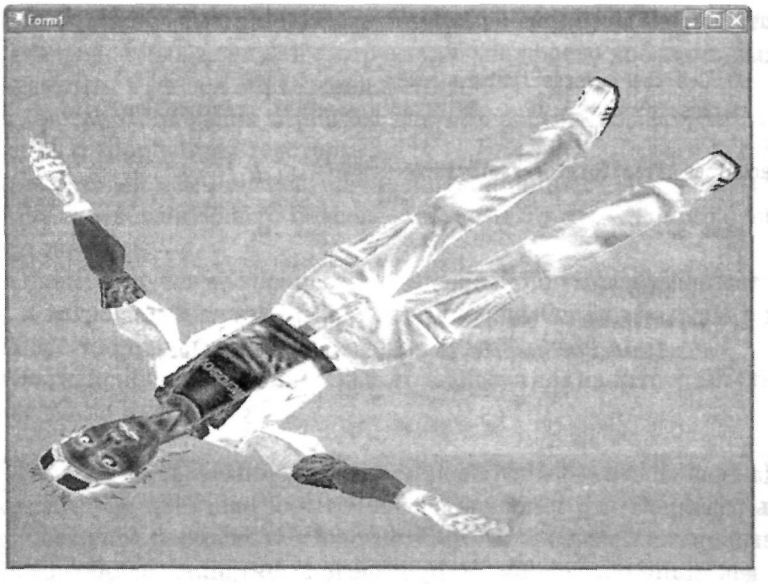


Рис. 11.4. Рендеринга mesh-объекта с обращенными цветами

Краткие выводы

В этой главе быши рассмотрены основные особенности языка HLSL. включая.

- Использование программируемого конвейера.
- Преобразование вершин с помощью шейдеров.
- Использование пиксельных шейдеров.

В следующей главе мы рассмотрим более совершенные методы языка HLSL.

Глава 12. Использование языка шейдеров HLSL

В предыдущей главе мы ввели понятие программируемого конвейера и рассмотрели его возможности в сравнении с непрограммируемым конвейером.

В этой главе мы рассмотрим особенности использования языка шейдеров HLSL для создания и отображения на экране более реалистичных картинок, нежели те, которые мы получали ранее, используя непрограммируемый конвейер.

Мы коснемся следующих вопросов.

- Простая анимация вершины.
  Простая цветная анимация.
- Объединение цвета текстуры с цветами поверхности.
- Световые модели с текстурами.
- Различие между повершинным per-vertex и попиксельным per-pixel наложением света.

Использование простых формул для моделирования анимации

Для примера мы возьмем цилиндр, рассмотренный нами в предыдущей главе, затем слегка изменим написанную для него программу, чтобы продемонстрировать возможность простой анимации с помощью программируемого конвейера. Мы попробуем отобразить качающийся объект.

Как только мы загрузили код из предыдущего примера, первое, что нам необходимо сделать, — это выбрать сферу вместо цилиндра. Это вовсе не означает, что мы не можем работать с цилиндром, просто сфера лучше смотрится. Замените код создания цилиндра на следующий:

```
// Create our sphere
mesh = Mesh.Sphere(device, 3.0f, 36, 36);
```

Поскольку размер нашей сферы больше, чем у вышеупомянутого цилиндра, необходимо изменить матрицу вида. Для этого нам необходимо переместить камеру назад, описав эту процедуру следующим образом:

```
viewMatrix = Matrix.LookAtLH(new Vector3(0,0, 9.0f), new Vector3(),
   new Vector3(0,1,0));
```

Теперь, запуская приложение, мы должны увидеть монотонно вращающуюся сферу, освещенную единственным направленным источником

света. Для создания эффекта анимации попробуем в коде шейдера заменить строку, описывающую преобразование, следующим кодом:

```
// Store our local position
float4 tempPos = Pos;
// Make the sphere 'wobble' some
tempPos.y += cos(Pos + (Time * 2.0f));
// Transform our position
Out.pos = mulftempPos, WorldViewProj) ;
```

Обратите внимание, что переменная Time используется уже неоднократно в наших примерах. Для данного примера необходимо переопределить эту переменную, добавив ее к коду шейдера:

```
float Time = 0.0f;
```

Кроме того, необходимо перезаписывать эту переменную при каждой смене кадра, поэтому в главном коде, в конце метода UpdateWorld, добавьте следующий код:

```
effect.SetValue("Time", angle);
```

Теперь опишем наши последние действия. Вначале исходное положение (входной параметр) вершины сохраняется как переменная. Затем компонента Y этой переменной модулируется путем добавления косинуса от суммы текущего местоположения вершины и значения Time*2.0f. Так как косинус меняется в пределах от -1 до +1, функция будет периодически повторятся. Переменная Time влияет на результат анимации в большей степени, нежели значение расположения Pos.

Можно добавить еще одно новшество, например, переменную анимации для цвета. Замените код инициализации Out.diff следующим:

```
// Set our color
Out.diff = sin(Pos + Time);
```

Очевидно, данный код аналогичен только что описанному методу вершинной анимации.

Единственное различие — использование функции синуса, который, судя по качеству отображаемой на экране картинки, предпочтительнее, чем косинус. Использование языка HLSL позволяет быстро применять различные способы отображения картинки без существенных изменений основного кода.

Объединение цветов текстуры с цветами поверхности

На практике встречается не так много приложений, использующих простые псевдослучайные цвета. Как правило, модели создаются или с помощью сложных текстур, или из нескольких текстур. Допустим, имеется сценарий, где необходимо смешать или наложить две или более текстуры. Бесспорно, это можно было бы сделать и на непрограммируемом конвейере. Но допустим, мы используем объект, похожий на «tiny.x», который мы уже неоднократно использовали в нашей книге. Данная модель использует только одну текстуру. Теперь попробуем для этой модели реализовать две текстуры, объединяя цвет текстуры с цветом поверхности (иногда используется термин интерполяция текстуры).

Для этого используем пример текстуры объекта, написанный в предыдущей главе. Попробуем добавить вторую текстуру, смешать эти текстуры и выдать окончательный цвет.

ПРЕДЕЛ ДОСТУПНЫХ КОМАНД ДЛЯ ПИКСЕЛЬНОГО ШЕЙДЕРА

Поскольку мы будем манипулировать пиксельным шейдером, необходимо обратить внимание на то, что количество доступных команд, которые могут быть использованы в шейдерах версии 1.1, чрезвычайно ограничено. Например, для шейдеров версий не старше 1.4 мы ограничены 12-ю командами в пределах всей программы. Для версии 1.4 мы имеем уже 28 команд и до 8 констант. Версия шейдера 2.0 и выше может выполнять гораздо более сложные операции с достаточным количеством команд и используемых констант. Вы можете заранее выяснить возможности вашей графической карты, чтобы определить, какие версии она поддерживает.

Зная теперь эти ограничения, будем создавать метод объединения двумя способами. Первый использует очень простой код шейдера 1.1, поддерживаемый практически всеми платами, второй — более сложный, для версии 2.0, с соответствующим ограничением на выбор платы.

Естественно, мы начнем с версии 1.1, как с более простой. Перед написанием соответствующего кода необходимо внести некоторые изменения. В первую очередь нужно определить и создать вторую текстуру. Добавьте следующую переменную для этой текстуры в ваш класс:

```
private Texture skyTexture;
```

Поскольку загружаемая нами модель имеет один материал и одну текстуру, можно удалить ссылку на использование дополнительных массивов материалов и текстур. Код, включенный в CD диск, уже включает в себя это изменение, поэтому на этом останавливаться не будем.

Итак, мы создаем или загружаем вторую текстуру, чтобы затем объединить ее с первой. В данном примере мы копируем текстуру skybox_top.JPG, которая находится в папке «application» DirectX SDK (можно в дальнейшем при желании использовать и другие текстуры). Как только мы выбрали вторую текстуру, необходимо ее создать. Наилучшим образом подойдет метод LoadMesh, которым была создана и первоначальная текстура. Исправьте строку создания текстуры:

```
// We have a texture, try to load it
meshTexture = TextureLoader.FromFile(device, @"..\..\" +
    mtrl[i].TextureFilename);
skyTexture = TextureLoader.FromFile(device, @"..\..\skybox_top.JPG");
```

Если вы выбрали другой файл текстуры, необходимо просто заменить имя файла в листинге. Теперь напишем код, позволяющий интерполировать имеющиеся текстуры. Добавьте следующие переменные к вашему коду шейдера:

```
float Time;
Texture meshTexture;
Texture skyTexture;
sampler TextureSampler = sampler_state { texture = <meshTexture>;
   mipfilter = LINEAR; };
sampler SkyTextureSampler = sampler_state { texture = <skyTexture>;
   mipfilter = LINEAR; };
```

Переменная Time использовалась нами достаточно часто, поэтому не требует разъяснений, а вот две переменные текстуры являются новыми. Используемая здесь выборка текстур с помощью процедуры сэмплирования более предпочтительна, нежели вызов SetTexture, который мы использовали раньше. По этой причине вызов SetTexture убирается из нашего приложения.

Две новые переменные являются внутренними переменными языка шейдеров и служат для определения того, как текстура сэмплирована. Таким образом, каждая сэмплируемая текстура основывается на текущей текстуре и использует линейный фильтр mipmap. Мы бы могли также включить фильтры растяжения или сжатия, но для данного случая фильтра mipmap достаточно.

Прежде чем использовать переменные текстурирования, необходимо определить их в программе инициализации основного кода , для этого добавьте следующий код после вызова загрузки mesh-объекта:

```
effect.SetValue("meshIexture", meshTexture);
effect.SetValue("skyTexture", skyTexture);
```

Таким образом, после загрузки mesh-объекта создаются наши текстуры. Однако, даже после этого наша модель имеет только один единственный набор координат текстуры. Используйте следующий код вершинного шейдера:

```
// Transform our coordinates into world space
void TransformVl_l(
    in float4 inputPosition : POSITION,
    in float2 inputTexCoord : TEXCOORDO,
    out float4 outputPosition : POSITION,
    out float2 outputTexCoord : TEXCOORDO,
    out float2 outputSecondTexCoord : TEXC00RD1
    )
{
    // Transform our position
    outputPosition = mul(inputPosition, WorldViewProj);
    // Set our texture coordinates
    outputTexCoord = inputTexCoord;
    outputSecondTexCoord = inputTexCoord;
}
```

Понятно, что здесь вводятся местоположение и набор координат отдельной текстуры. Однако, на выходе программа не просто возвращает преобразованное местоположение текстуры, но также пересылает входную координату текстуры к обоим наборам выходных координат. По существу, код дублирует координаты текстуры для каждой вершины поверхности. Теперь мы можем заменить код пиксельного шейдера на следующий:

```
void TextureColorVl_l (
    in float4 P : POSITION,
    in float2 textureCoords : TEXCOORDO,
    in float2 textureCoords2 : TEXC00RD1,
    out float4 diffuseColor : COLORO)
    {
        // Get the texture color
        float4 diffuseColorl = tex2D(TextureSampler, textureCoords);
        float4 diffuseColor2 = tex2D(SkyTextureSampler, textureCoords2);
        diffuseColor = lerp(diffuseColorl, diffuseColor2, Time);
    };
```

Здесь процедура принимает месторасположение и наборы координат текстур, возвращенные из подпрограммы вершинного шейдера, и выводит цвет, которым должен быть отображен данный пиксел. В этом случае

мы сэмплируем каждую из двух загруженных текстур (с идентичными координатами текстур), а затем выполняем линейную интерполяцию (встроенная функция lerp). Значение переменной Time определяет то, как долго мы видим каждую текстуру. Пока мы не задавали данную переменную в нашем приложении. Можем сделать это в методе mesh drawing следующим образом:

```
effect.SetValue("Time",  (float)Math.Abs(Math.Sin(angle)));
```

Напрашивается вопрос, почему мы выполняем «математику» именно здесь, а не в программе шейдера? Конечно, было бы предпочтительнее возложить данные операции на графическую карту, чем на центральный процессор, но мы об этом уже писали. Имеются в виду ограничения, связанные с версией 1.1 используемого шейдера и нашей картой.

Итак, перед запуском приложения с нашими текстурами осталось выполнить одну замену. Поскольку имена функций для вершинного и пиксельного шейдеров были изменены, необходимо заменить используемую нами технику:

```
technique  TransformTexture
{
   pass PO
   {
      //  shaders
      VertexShader = compile vs_l_l TransformVl_l();
      PixelShader = compile ps_l_l TextureColorVl_l();
   I
1
```

Название самой техники не изменилось, и теперь мы можем запустить наше приложение. Обратите внимание на то, что текстурируемая при запуске модель «смешивается» с текстурой неба Sky Texture (полагая, что мы использовали эту текстуру) и затем снова объединяется с исходной текстурой. Это повторяется до тех пор, пока приложение не будет закрыто.

Теперь предположим, что ваша графическая карта поддерживает пиксельный шейдер версии 2.0. Мы должны переписать код таким образом, чтобы иметь возможность работать с обеими версиями пиксельных шейдеров 1.1 и 2.0. Чтобы узнать, поддерживает ли плата версию 2.0, необходимо добавить переменную к основному коду:

```
private bool canDo2_0Shaders = false;
```

Значение «false» предполагает, что плата не поддерживает версию 2.0 первоначально. После того как мы создали устройство в методе инициа-

лизации, необходимо выяснить, может ли шейдер 2.0 поддерживаться самим устройством. Этот вызов необходимо определить непосредственно перед командой Effect:

```
canDo2_0Shaders = device.DeviceCaps.PixelShaderVersion >= new Version(2, 0);
```

Поскольку мы добавляем вторую технику для шейдера, необходимо, основываясь на более продвинутой модели шейдера, определить, какая именно техника будет являться базовой. Для этого перепишите запись effect.Technique:

```
effect.Technique = canDo2JShaders ? "TransformTexture2_0" :
"TransformTexture";
```

В этом случае, если модель 2.0 поддерживается, приложение будет использовать эту технику, в противном случае будет использоваться техника, которую мы уже написали. Теперь перепишем строку, которая определяет переменную Time:

```
if (canDo2_0Shaders)
{
   effect.SetValue("Time", angle);
}
   else
{
   effeet.SetValue("Time", (float)Math.Abs(Math.Sin (angle)));
}
```

Обратите внимание, что с более совершенной программой шейдера и соответствующей графической картой выполнение математических операций будет возложено на графическую карту, и мы воспользуемся всеми преимуществами данной конфигурации. Теперь нам необходимо добавить новый код шейдера, который позволит нам выполнить приложение. Добавьте код, приведенный в листинге 12.1, к вашему коду HLSL.

Листинг 12.1. Процедура объединение текстур на языке шейдера версии 2.0.

```
// Transform our coordinates into world space
void TransformV2_0(
   in float4 inputPosition : POSITION,
   in float2 inputTexCoord : TEXCOORDO,
   out floats outputPosition : POSITION,
   out float2 outputTexCoord : TEXCOORDO
   )
(
```

```
    // Transform our position
    outputPosition = mul(inputPosition, WorldViewProj);
    II Set our texture coordinates
    outputTexCoord = inputTexCoord;
)
void TextureColorV2_0(
in float4 P : POSITION,
in float2 textureCoords : TEXCOORDO,
out float4 diffuseColor : COLORO)
(
    // Get the texture color
    float4 diffuseColorl = tex2D(TextureSampler, textureCoords);
    float4 diffuseColor2 = tex2D(SkyTextureSampler, textureCoords);
    diffuseColor = lerp(diffuseColorl, diffuseColor2, abs(sin(Time)));
};
technique TransformTexture2_0
{
    pass PO
    {
        // shaders
        VertexShader = compile vs_l_l TransformV2_0();
        PixelShader = compile ps_2_0 TextureColorV2_0() ;
    }
}
```

Наша программа стала намного проще. Вместо дублирования координат текстуры код просто преобразовывает местоположение и пересылает первоначальный набор координат. Нет ничего похожего на программы вершинных шейдеров (vertex program).

Пиксельный шейдер вообще выглядит намного проще. Здесь нет необходимости в использовании двух наборов координат текстур, вместо этого смешивание двух различных текстур происходит с помощью одного и того же набора текстурных координат. В более старых версиях пиксельных шейдеров (до версии 2.0) мы могли читать координаты текстуры только один раз, и процедура смешивания двух текстур с помощью одного набора координат вызвала бы два считывания, а это было невозможно при наличии старой версии шейдера. Следует также заметить, что для определения уровня интерполяции (функция lerp) благодаря тому, что математика выполняется в коде шейдера, используется одна и та же формула. Единственные различия в данной технике — это имена функций и то, что пиксельный шейдер компилируется с помощью ps_2_0.

Текстуры освещения

В предыдущей главе мы обсуждали простое направленное освещение. На том этапе цилиндр не текстурировался, и расчет освещения вы-

поднялся только вершинными шейдерами. Мы можем легко осуществить те же операции, используя пиксельные шейдеры.

Подобно предыдущему примеру, мы выберем простой mesh-объект из предыдущей главы в качестве отправной точки. В этом разделе возьмем готовый пример и добавим цветное освещение к сцене в нашем пиксельном шейдере.

В качестве первого шага нам необходимо добавить некоторые объявления к коду шейдера:

```
float4x4 WorldMatrix : WORLD;
floats DiffuseDirection;
Texture meshTexture;
sampler TextureSampler = sampler^state { texture = <meshTexture>; mipfilter
= LINEAR; };
struct VS_OUTPUT
{
   float4 Pos : POSITION;
   float2 TexCcord : TEXC0ORD3;
   float3 Light : TEXC00RD1;
   float3 Normal : TEXC00RD2;
};
```

Обратите внимание, что мы не просто сохраняем комбинированную матрицу вида и проекции как раньше, а имеется другая переменная для записи непосредственно мировой матрицы. Мы поймем необходимость этого, как только обсудим данный код шейдера. Также имеется переменная, отвечающая за направление света. Как и в предыдущем примере, имеется текстура и состояние сэмплера для чтения текстуры.

Выходная структура данной процедуры несколько отличается. Пересылаются не только местоположение и координаты текстуры, пересылаются также два новых значения координат текстуры. В действительности они не являются координатами текстуры как таковыми, они используются только для передачи необходимых данных из вершинного шейдера в пиксельный шейдер.

Мы должны также переопределить эти значения в методе рисования объекта после вызова строки с переменной WorldViewProj:

```
effect.SetValue("WorldMatrix", worldMatrix);
effect.SetValue("DiffuseDirection", new Vector4(0, 0, 1, I));
```

А также устанавливаем переменную текстуры, добавляя в метод инициализации следующий код после процедуры загрузки mesh-объекта (и текстуры):

```
effect.SetValue("meshTexture", meshTexture);
```

Теперь все готово к тому, чтобы мы могли переписать наш код пиксельного шейдера и отобразить текстуру освещения. Осталось выбрать цвет освещения (любой кроме белого, для наглядности) и проверить вершинный шейдер:

```
// Transform our coordinates into world space
VS_OUTPUT Transform)
   float4 inputPosition : POSITION,
   float3 inputNormal : NORMAL,
   float2 inputTexCoord : TEXCOORDO


   //Declare our output structure
   VS_0UTPUT Out = (VS_OUTPUT)0;
   // Transform our position
   Out.Pos = mul(inputPosition, WorldViewProj);
   // Store our texture coordinates
   Out.TexCoord = inputTexCoord;
   // Store our light direction
   Out.Light = DiffuseDirection;
   // Transform the normals into the world matrix and normalize them
   Out.Normal = normalize(mul(inputNormal, WorldMatrix));
   return Out;
}
```

Начало этой программы шейдера для нас достаточно ясно, хотя в ней имеется значительное число входных параметров. Вначале преобразуется местоположение и записываются координаты текстуры. Затем параметр направления света «light direction» размещается во второй набор координат текстуры. Обратите внимание на то, что тип параметра направления света — float4, тогда как тип координат текстуры — float3.

В третьем наборе координат текстуры необходимо сохранить данные о нормалях. Прежде чем это сделать, необходимо преобразовать нормали в мировую систему координат. Без этого расчет освещения будет выполняться некорректно, нельзя выполнять математические операции над объектами в различных системах координат. Выполнив все необходимые действия с освещением и текстурами, мы можем добавить следующий код пиксельного шейдера:

```
floats TextureColor(
float2 textureCoords : TEXCOORDO,
float3 lightDirection : TEXC00RD1,
float3 normal : TEXC00RD2) : COLORO
{
   // Get the texture color
   float4 textureColor = tex2D(TextureSampler, textureCoords);
```

```
    // Make our diffuse color purple for now
    float4 diffuseColor = (1.0f, 0.0f, 1.0f, 1.0f);
    // Return the combined color after calculating the effect of
    // the diffuse directional light
    return textureColor * (diffuseColor * saturate(dot(lightDirection,
    normal)));
};
```

Как вы видите, в качестве входных параметров программа пиксельного шейдера использует все наборы координат текстуры, возвращенные программой вершинного шейдера, и определяет цвет этого пиксела. Первым шагом является выбор текущей текстуры. Затем устанавливается цвет освещения (в данном случае — фиолетовый). Чтобы рассчитать освещение в пикселе, необходимо взять скалярное произведение направления освещения и нормали, как это уже делалось раньше для вершинного шейдера. Встроенный метод saturate (насыщение) удерживает возвращаемое значение яркости в диапазоне от нуля до единицы, обеспечивая выравнивание чрезмерно ярких или чрезмерно темных цветов. Окончательно результирующий цвет получается умножением текущего цвета текстуры на диффузные компоненты освещения.

Несмотря на то, что этот пример чрезвычайно прост, он заслуживает внимания и является достаточно общим случаем. Запуская приложение, мы должны увидеть отображаемую модель с фиолетовым оттенком.

Моделирование световых бликов

Другим типом освещения, который мы еще не обсуждали, является «specular highlighting» или моделирование световых бликов на поверхности. Данный метод позволяет отобразить объект на экране более реалистично. Мы можем реализовать данный тип освещения с помощью непрограммируемого конвейера, однако при этом эти подсветки будут рассчитаны повершинно. В этой главе мы покажем, как можно реализовать световые блики, используя программируемый конвейер.

Мы будем опираться на более раннюю секцию кода и за основу возьмем пример кода для освещенного цилиндра, написанный в предыдущей главе. Данный код использует метод для случая диффузного освещения, его мы оставим для сравнения, добавив соответствующий алгоритм для новой модели освещения.

Для большей наглядности мы выберем вместо цилиндра объект teapot. Для этого перепишите код загрузки цилиндра, заменив название загружаемого объекта:

```
// Load our mesh
mesh = Mesh.Teapot(device);
```

Поскольку этот пример будет демонстрировать различные эффекты, следует позаботиться о возможности переключения между ними, и, кроме того, было бы неплохо добавлять соответствующие текстовые комментарии. Поэтому необходимо определить переменную шрифта:

```
// Out font
private Direct3D.Font font = null;
```

Инициализируем переменную шрифта сразу после создания объекта teapot:

```
//Create our font
font = new Direct3D.Font(device, new System.Drawing.Font("Arial", 12.Of));
```

Теперь необходимо добавить константы и переменные, которые будут задействованы в коде пиксельного шейдера:

```
float4x4 WorldViewProj : WORLDMEWPROJECTION;
float4x4 WorldMatrix : WORLD;
float4 DiffuseDirection;
float4 EyeLocation;
// Color constants
const float4 MetallicColor = { 0.8f, 0.8f, 0.8f, 1.Of };
const float4 AmbientColor = { 0.05f, 0.05f, 0.05f, 1.Of };
```

Для преобразования вершин здесь используются мировая матрица, матрица вида и матрица проекции. Мировая матрица будет еще раз использована для преобразования нормалей к вершинам. Значение DiffuseDirection позволит приложению определять направление света быстрее, чем при использовании шейдера в предыдущих примерах. И последняя переменная EyeLocation — положение глаза наблюдателя. Световые блики рассчитываются, исходя из отражения света между нормалью к поверхности и глазом.

Обратите внимание на то, что в нашей подпрограмме имеются две объявленные константы. Цвет MetallicColor выбирается, благодаря свойствам металлической поверхности и падающего на него диффузного освещения. Вторая объявленная константа — общее освещение AmbientColor. Данный параметр включен для полноты математических операций при расчете освещения.

Для данного примера выходная структура повершинного освещения per-vertex рассматривает и задает местоположение и цвет каждой вершины. Мы можем объявить ее следующим образом:

```
struct VS_OOTPUT_PER_VERTEX
1
   float4 Position : POSITION;
   float4 Color : COLORO;
I;
```

Прежде чем написать код шейдера для моделирования световых бликов, необходимо переписать код шейдера для диффузного освещения.

Таким образом мы создадим отдельные шейдеры для каждой световой модели. Замените программу для диффузного освещения следующим кодом:

```
VS_OUTPUT_PER_VERTEX  TransformDiffuse(
   floats inputPosition : POSITION,
   float3 inputNormal : NORMAL,
   uniform bool metallic
   )
{
   //Declare our output structure
   VS_0UTPUT_PER_VERTEX Out = (VS_OUIPUT_PER_VERTEX)0;
   // Transform our position
   Out.Position = mul(inputPosition, WorldViewProj);
   // Transform the normals into the world matrix and normalize them
   float3 Normal = normalize(mul(inputNormal, WorldMatrix));
   // Make our diffuse color metallic for now
   float4 diffuseColor = MetallicColor;
   if(!metallic)
   diffuseColor.rgb = sin(Normal + inputPosition);
   // Store our diffuse component
   float4 diffuse = saturate(dot(DiffuseDirection, Normal));
   // Return the combined color
   Out.Color = AmbientColor + diffuseColor * diffuse;
   return Out;
}
```

Обратите внимание, что в шейдере появился новый входной параметр, булева переменная, объявленная с атрибутом «uniform». Модификатор «uniform» позволяет приложению Direct3D обработать эту переменную как константу, которая не может быть изменена в течение вызова рисования draw.

При запуске данного шейдера преобразуется местоположение и затем нормаль, а для диффузного освещения выбирается объявленный ранее цвет «metallic».

Также присутствует оператор, который мы еще не рассматривали, а именно, управление потоком данных (flow control). Язык HLSL поддерживает несколько механизмов управления потоками данных, включая знакомые нам условные операторы «if», операторы циклов, прерывание цикла «while loop» и т. д. Каждый из механизмов управления потоками данных имеет свой синтаксис, подобный эквивалентным операторам в C#.

УПРАВЛЕНИЕ ПОТОКАМИ ДАННЫХ НА СТАРЫХ ВЕРСИЯХ ШЕЙДЕРОВ

Более старые версии шейдеров не поддерживают управление потоками данных или переходы.

Компилятор HLSL может раскручивать цикл или выполнять все ветви условных операторов. Это необходимо знать, чтобы при написании сложных шейдеров не достичь предельного количества операторов. Возьмем простой цикл:

```
for(int i = 0; i<10; i++)
{
   pos.y += (float)i;
}
```

Даже при том, что здесь имеется только один оператор, данный код шейдера инициировал бы по крайней мере выполнение 20 команд. И если бы шейдер имел ограничение в 12 команд (например, шейдер версии 1.1), этот простой цикл не был бы откомпилирован.

Итак, вернемся к нашей программе. Если переменная metallic истинна, то сохраняется металлический цвет освещения.

Если ложное, цвет будет переключен на цвет «animating», аналогично первому примеру в этой главе.

Теперь, имея два различных типа цвета, которые могут использоваться в этой программе шейдера, мы добавляем две техники:

```
technique TransformSpecularPerVertexMetallic
{
   pass P0
   {
      // shaders
      VertexShader = compile vs_1_1 TransformSpecular(true) ;
      PixelShader = NULL;
   }
}
technique TransformSpecularPerVertexColorful
{
   pass P0
   {
      // shaders
      VertexShader = compile vs_1_1 TransformSpecular(false);
      PixelShader = NULL;
   }
}
```

Следует обратить внимание, что единственное различие между этими двумя техниками (помимо разных названий) — значение, которое пересылается в код шейдера.

Далее, поскольку названия используемых техник изменились, необходимо переписать основной код:

```
effect.Technique = "TransformDiffusePerVertexMetallic";
```

Далее необходимо добавить новую программу шейдера для моделирования световых бликов с учетом диффузного освещения и использования переходов за область оператора в программе. Добавьте метод, приведенный в листинге 12.2, к вашему коду HLSL.

Листинг 12.2. Преобразования и использование световых бликов.

```
VS_OUTPUT_PER_VERTEX TransformSpecular (
   float4 inputPosition : POSITION,
   float3 inputNormal : NORMAL,
   uniform bool metallic
   )
{
   //Declare our output structure
   VS_OUTPUT_PER_VERTEX Out = (VS_OUTPUT_PER_VERTEX)0;
   // Transform our position
   Out.Position = mul(inputPosition, WorldViewProj);
   // Transform the normals into the world matrix and normalize them
   float3 Normal = normalize(mul(inputNormal, WorldMatrix));
   // Make our diffuse color metallic for now
   float4 diffuseColor = MetallicColor;
   // Normalize the world position of the vertex
   float3 worldPosition = normalize(mul(inputPosition, WorldMatrix));
   // Store the eye vector
   float3 eye = EyeLocation - worldPosition;
   // Normalize our vectors
   float3 normal = normalize(Normal);
   float3 light = normalize(DiffuseDirectipn);
   float3 eyeDirection = normalize(eye);
   if([metallic)
      diffuseColor.rgb = cos(normal + eye);
   // Store our diffuse component
   float4 diffuse = saturate(dot(light, normal));
   // Calculate specular component
   float3 reflection = normalize(2 * diffuse * normal - light);
   floats specular = pow(saturate(dot(reflection, eyeDirection)), 8);
```

```
    // Return the combined color
    Out.Color = AmbientColor + diffuseColor * diffuse + specular;
    return Out;
}
```

Это самая длинная программа шейдера в этой книге из рассмотренных до настоящего момента. Начало аналогично предыдущему случаю, когда мы преобразовывали местоположение и нормаль и затем устанавливали диффузное освещение для константы цвета «metallic». После этого в программе сохраняется местоположение каждой вершины в мировых координатах, поскольку наблюдатель расположен в пространстве мировых координат, а вершины — в пространстве модели. И, поскольку при вычислении световых бликов будет использоваться направление от наблюдателя, все значения должны находиться в одной системе координат. -

Далее каждый из векторов нормализуется к значению 1.Of, после чего проверяется булева переменная, и пересчитывается значение цвета с учетом нового нормализованного вектора и сохраненного диффузного освещения, входящих в формулу расчета, таким образом, моделируется случайный цвет анимации и окончательно рассчитывается модель световых бликов. Для получения дополнительной информации по формуле расчета освещения см. документацию DirectX SDK.

После того как рассчитаны компоненты световой модели, программа шейдера возвращает результирующий цвет, используя те же самые математические формулы для освещения. После того как мы написали тендеры для моделирования световых бликов, можно добавить следующие процедуры обращения к ним:

```
technique  TransformSpecularPerVertexMetallic
{
   pass PO
   {
      // shaders
      VertexShader = compile vs__l_l TransformSpecular(true);
      PixelShader = NULL;
   }
}
technique  TransformSpecularPerVertexColorful
{
   pass PO
   {
      // shaders
      VertexShader = compile vs_l_l TransformSpecular(false);
      PixelShader = NULL;
   }
   }
```

Перед запуском приложения необходимо переписать основной код, чтобы вызвать программу шейдера для моделирования световых бликов:

```
effect.Technique = "TransformSpecularPerVertexMetallic";
```

На рис.12.1 изображен вращающийся, освещенный светом чайник.

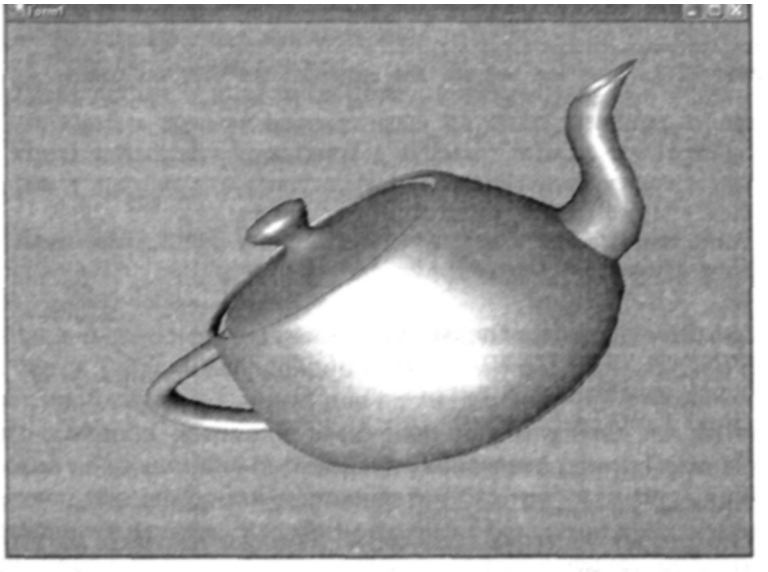


Рис. 12.1 Модель чайника использующая повершииное (per-vertex) моделирование световых бликов

ИСПОЛЬЗОВАНИЕ ПОПИКСЕЛЬНОГО (PER-PIXEL) МОДЕЛИРОВАНИЯ СВЕТОВЫХ БЛИКОВ

Легко заметить, что освещенный солнцем чайник выглядит гораздо реалистичнее, если мы используем световые блики. Тем не менее, может показаться, что повершинное освещение, рассчитываемое в этой программе, не так гладко освещает изогнутую поверхность заварочного чайника, как хотелось бы. Чтобы получить более мягкую игру света, можно рассчитать освещение, используя попиксельное моделирование освещения.

Поскольку вычисление освещения требует большего количества команд, чем позволяет пиксельный шейдер версии 1.1, необходимо убедиться, что ваше приложение поддерживает по крайней мере шейдеры версии не ниже 2.0. Чтобы проверить эту «логику» перепишите основной код:

```
if ((hardware.VertexShaderVersion >= new Version(1, 1)) &&
    (hardware.PixelShaderVersion >= new Version(2, 0)))
```

Нам все еще понадобится вершинный шейдер для преобразования координат и передачи данных, которые будут использоваться в пиксельном шейдере. Теперь добавьте программу шейдера, приведенную в листинге 12.3.

Листинг 12.3. Программа шейдера для моделирования световых бликов.

```
struct VS_OUTPUT_PER_VERTEX_PER_PIXEL
{
   float4 Position : POSITION;
   float3 LightDirection : TEXCOORDO;
   float3 Normal : TEXCOORD1;
   float3 EyeWorld : TEXCOORD2;
};
// Transform our coordinates into world space
VS_OUTPUT_PER_VERTEX_PER_PIXEL Transform(
   float4 inputPosition : POSITION,
   float3 inputNoraal : NORMAL
   )
{
   //Declare our output structure
   VS_OUTPUT_PER_VERTEX_PER_PIXEL Out = (VS_OUTPUT_PER_VERTEX_PER_PIXEL)
   // Transform our position
   Out.Position = mul(inputPosition, WorldViewProj);
   // Store our light direction
   Out.LightDirection = DiffuseDirection;
   // Transform the normals into the world matrix and normalize them
   Out.Normal = normalize(mul(inputNormal, WorldMatrix));
   // Normalize the world position of the vertex
   float3 worldPosition = normalize(mul(inputPosition, WorldMatrix));
   // Store the eye vector
   Out.EyeWorld = EyeLocation - worldPosition;
   return Out;
}
float4 ColorSpecular(
float3 lightDirection : TEXCOORDO,
float3 normal : TEXC00RD1,
float3 eye : TEXC00RD2,
uniform bool metallic) : COLORO
```

```
    // Make our diffuse color metallic for now
    floats diffuseColor = MetallicColor;
    if(!metallic)
    diffuseColor.rgb = cos(normal + eye);
    // Normalize our vectors
    float3 normalized = normalize(normal);
    float3 light = normalize(lightDirection);
    float3 eyeDirection = normalize(eye);
    // Store our diffuse component
    float4 diffuse = saturate(dot(light, normalized));
    // Calculate specular component
    float3 reflection = normalize(2 * diffuse * normalized- light);
    float4 specular = pow(saturate(dot(reflection, eyeDirection)), 8);
    // Return the combined color
    return AmbientColor + diffuseColor * diffuse + specular;
};
```

Написанная программа напоминает код для повершинной обработки света. Вершинный шейдер выполняет преобразования и сохраняет необходимые данные для последующей программы пиксельного шейдера, которая в свою очередь выполняет все математические операции для определения конечного цвета. Поскольку попиксельные операции выполняются быстрее чем повершинные, полученная в результате картинка будет более мягкой и реалистичной. Добавьте следующую методику к вашему файлу шейдера:

```
technique TransformSpecularPerPixelMetallic
{
    pass PO
    {
        // shaders
        VertexShader = compile vs_1_1 Transform!);
        PixelShader = compile ps_2_0 ColorSpecular(true);
    }
    }
```

Переключите приложение, чтобы можно было сравнить результаты использования двух методик.

На **рис.12.2** изображена более реалистичная картинка вращающегося освещенного чайника.

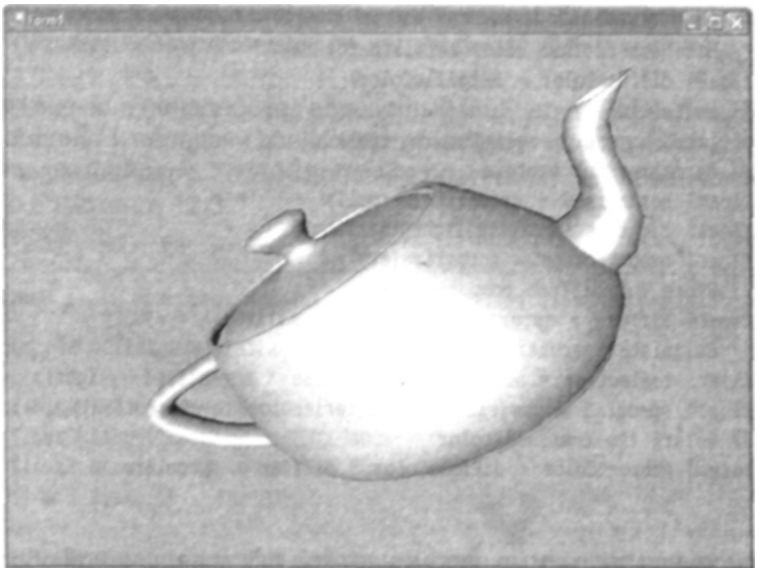


Рис. 12.2. Модель чайника, использующая попиксельное (per-pixel) моделирование световых бликов

Пример, включенный в CD диск, позволяет плавное переключение между попиксельным и повершинным исполнением программы, а также позволяет переключать между металлическим и полноцветным освещением поверхности.

Краткие выводы

В этой главе мы рассмотрели следующие вопросы.

- **Простая анимация вершины и цветная анимация.**
- **Объединение цвета текстуры с цветами поверхности.**
- **Более совершенные модели освещения.**

В следующей мы обсудим создание анимации в Управляемом DirectX

Глава 13. Рендеринг скелетной анимации

Как правило, в современных играх мы наблюдаем непрерывные сглаженные динамические перемещения объектов. Элементы полностью анимированы, при анимации используется система захвата движения объекта (motion capture studio). Скелетная анимация используется при моделировании различных движений объекта или его частей. Помимо сложных мультипликаций имеются также другие, более простые типы анимации, имеющие дело главным образом с масштабированием, вращением и перемещением. В этой главе мы рассмотрим рендеринг mesh-объектов с анимацией данных, включая.

- Загрузку иерархической системы фреймов.
- Формирование скелетных данных.
  Рендеринг каждого фрейма.
- Оптимизацию времени анимации.
  Использование индексированной анимации для улучшения быстродействия.

Создание иерархической системы фреймов

Большинство mesh-объектов (даже не имеющих встроенной анимации) обладают некоторой иерархией: рука, приложенная к груди, палец, приложенный к руке и т. д. Данные иерархии можно загружать и удерживать с помощью библиотеки расширений Direct3D.

Чтобы сохранять эту информацию, существует два абстрактных класса, которые поддерживают иерархию: класс Frame и класс MeshContainer. Каждый фрейм может содержать родственные или дочерние фреймы. Каждый фрейм может также содержать нулевой или свободный указатель zero к контейнерам класса MeshContainer.

Как обычно, начнем написание программы для Direct3D с создания нового проекта, определения переменных, окон и т. п. Объявите следующие переменные в приложении:

```
private AnimationRootFrame rootFrame;
private Vector3 objectCenter;
private float objectRadius;
private float elapsedTime;
```

Структура AnimationRootFrame содержит корневой фрейм дерева класса анимации. Следующие две переменные будут содержать центр загружаемого объекта mesh и радиус сферы, ограничивающей объект. Это позволит позиционировать камеру таким образом, чтобы захватывать всю

модель. Последняя переменная определяет время выполнения, которое будет использоваться для обновления анимации.

Поскольку различные объекты анимируются по-разному, необходимо создать новый производный класс из объекта иерархий AllocateHierarchy. Для этого добавьте к приложению следующий класс:

```
public class AllocateHierarchyDerived : AllocateHierarchy
{
   Forml app = null;
   /// <summary>
   /// Create new instance of this class
   /// </summary>
   /// <param name="parent">Parent of this class</param>
   public AllocateHierarchyDerived(Forml parent)
   {
      app = parent;
   }
}
```

Пока данный класс не задействован. Конструктор программ сохраняет его основную форму так, чтобы в дальнейшем использовать данный класс для вызова соответствующих методов, но на данном этапе приложение не будет его даже компилировать. Имеются два абстрактных метода, которые должны быть реализованы в этом классе еще до компиляции. Эти два метода используются для создания фреймов и контейнеров MeshContainer, позволяющих сохранить иерархию. Поскольку оба метода помечены как абстрактные, они не могут быть созданы автоматически, поэтому необходимо добавить новые версии производных классов к вашему приложению:

```
public class FrameDerived : Frame
{
   // Store the combined transformation matrix
   private Matrix combined = Matrix.Identity;
   public Matrix CombinedTransformationMatrix
   {
      get { return combined; } set { combined = value; }
   }
}
```

Таким образом, в дополнении к данным нормали, записанным во фрейме, каждый фрейм будет сохранять матрицу преобразований, скомбинированную из данных фрейма и данных всех его «родителей». Это может пригодиться в дальнейшем при формировании мировых матриц. Теперь необходимо включить производный класс mesh container, см. листинг 13.1.

Листинг 13.1. Добавление класса MeshContainer.

```
public class MeshContainerDerived : MeshContainer
{
   private Texture[] meshTextures = null;
   private int numAttr = 0;
   private int numlnfl = 0;
   private BoneCombination[] bones;
   private FrameDerived[] frameMatrices;
   private Matrix[] offsetMatrices;
   // Public properties
   public Texture[] GetTextures() { return meshTextures; }
   public void SetTextures(Texture[] textures) { meshTextures = textures; }
   public BoneCombination[] GetBones() { return bones; }
   public void SetBones(BoneCombination[] b) { bones = b; }
   public FrameDerived[] GetFrames() { return frameMatrices; }
   public void SetFrames(FrameDerived[] frames) { frameMatrices = frames; }
   public Matrix[] GetOffsetMatrices() { return offsetMatrices; )
   public void SetoffsetMatrices(Matrix[] matrices){offsetMatrices = matrices; }
   public int NumberAttributes {get{return numAttr;} set{numAttr = value;}}
   public int Numberlnfluences {get{return numInfl;}set{numInfl = value;}}
}
```

Каждый контейнер MeshContainer будет содержать информацию, включающую таблицу комбинаций каркасов, номера атрибутов, данные о матрицах влияний и матрицах смещения. Каждый параметр этих данных будет использоваться при рендеринге нашего анимационного объекта.

Создав необходимые производные классы, можно добавить абстрактные методы. Вначале добавьте перегрузку CreateFrame к производному классу allocate hierarchy:

```
public override Frame CreateFrame(string name)
{
   FrameDerived frame = new FrameDerived();
   frame. Name = name;
   frame.TransformationMatrix = Matrix.Identity;
   frame.CombinedTransformationMatrix = Matrix.Identity;
   return frame;
}
```

Данный код записывает название фрейма, объявляет матрицы преобразования и т. д. Родственные и дочерние фреймы, а также контейнеры MeshContainer будут заполняться автоматически за время выполнения приложения.

Для создания контейнера MeshContainer добавьте код, приведенный в листинге 13.2.

Листинг 13.2. Создание раздела MeshContainer.

```
public override MeshContainer CreateMeshContainer(string name,
   MeshData meshData, ExtendedMaterial[] materials,
   EffectInstance effectInstances, GraphicsStream adjacency,
   SkinInformation skinInfo)
{
   // We only handle meshes here
   if (meshData.Mesh == null)
      throw new ArgumentException();
   // We must have a vertex format mesh
   if (meshData.Mesh.VertexFormat == VertexFormats.None)
      throw new ArgumentException();
   MeshContainerDerived mesh = new MeshContainerDerived();
   mesh. Name = name;
   int numFaces = meshData.Mesh.NumberFaces;
   Device dev = meshData.Mesh.Device;
   // Make sure there are normals
   if ((meshData.Mesh.VertexFormat & VertexFormats.Normal) == 0)
   {
      // Clone the mesh
      Mesh tempMesh = meshData.Mesh.Clone(meshData.Mesh.Options.Value,
         meshData.Mesh.VertexFormat | VertexFormats.Normal, dev);
      meshData.Mesh = tempMesh;
      meshData.Mesh.ComputeNormals();
}
// Store the materials
mesh.SetMaterials(materials);
mesh.SetAdjacency(adjacency);
Texture[] meshTextures = new Texture[materials.Length];
// Create any textures
for (int i = 0; i < materials.Length; i++)
{
      if (materials[i].TextureFilename != null)
      {
         meshTextures[i] = TextureLoader.FromFile(dev, @"..\..\" +
            materials[i].TextureFilename);
      }
}
mesh.SetTextures(meshTextures);
mesh.MeshData = meshData;
// If there is skinning info, save any required data
if (skinInfo != null)
```

```
    mesh.SkinInformation = skinInfo;
    int numBones = skinInfo.NumberBones;
    Matrix[] offsetMatrices = new Matrix[numBones];
    for (int i = 0; i < numBones; i++)
        offsetMatrices[i] = skinInfo.GetBoneOffsetMatrix(i);
        mesh.SetOffsetMatrices(offsetMatrices) ;
        app.GenerateSkinnedMesh(mesh);
    }
    return mesh;
}
```

Это выглядит несколько сложнее, чем на самом деле. В первую очередь проверяется, поддерживает ли приложение данный объект. Очевидно, что при отсутствии объекта, уже включенного в структуру данных, параметр будет исключен (строка throw new ArgumentException). То же самое произойдет, если включенный в структуру объект не использует допустимый вершинный формат.

В случае, если данные объекта mesh достоверны, создается новый контейнер MeshContainer, куда наряду с устройством записываются имя и число поверхностей объекта. Код может также использовать дополнительную индивидуальную переменную для доступа к винформам устройства.

Далее идет более-менее знакомый нам фрагмент кода — проверяются данные нормали для объекта. Если данные отсутствуют, программа добавляет их к объекту. Затем сохраняются материалы и информация смежности «adjacency», а также создается массив данных текстур заданных материалов. Далее записывается соответствующая текстура в каждый член массива, который затем сохраняется в контейнере MeshContainer.

В конце проверяется наличие скелетной информации в контейнере MeshContainer, после чего информация сохраняется, и формируется массив из матриц смещения (для каждого каркаса). После этого из нашей формы вызывается пока еще не существующий метод GenerateSkinnedMesh. По этой причине приложение все еще не будет компилировать программу, необходимо добавить метод GenerateSkinnedMesh, приведенный в листинге 13.3, в наш основной класс.

Листинг 13.3. Создание каркасных объектов Skinned Meshes.

```
public void GenerateSkinnedMesh(MeshContainerDerived mesh)
 {
    if (mesh.SkinInformation == null)
        throw new ArgumentException() ;
    int numInfl = 0;
```

```
    BoneCombination[] bones;
    // Use ConvertToBlendedMesh to generate a drawable mesh
    MeshData m = mesh.MeshData;
    m.Mesh = mesh.SkinInformation.ConvertToBlendedMesh(m.Mesh,
    MeshFlags.Managed
        | MeshFlags.OptimizeVertexCache, mesh.GetAdjacencyStream(), out numInfl,
        out bones);
    // Store this info
    mesh.NumberInfluences = numInfl;
    mesh.SetBones (bones);
    // Get the number of attributes
    mesh.NumberAttributes = bones.Length;
    mesh.MeshData = m;
}
```

Данный метод никогда не вызывается, если не имеется никакой скелетной информации. Мы временно сохраняем данные объекта и используем метод ConvertToBlendedMesh, чтобы создать новый объект с поверхностной комбинацией весов и таблицей комбинаций каркасных элементов. Это — «корневой» метод для анимирования объекта. Наконец, мы сохраняем набор связей в этом объекте, таблицу комбинаций каркасных элементов и атрибуты.

Загрузка объектов с анимацией

Итак, в предыдущем параграфе была установлена основная иерархия объекта. Теперь нам необходимо записать некоторый код, чтобы фактически использовать ее. Как обычно, нам нужно инициализировать графическое устройство, используя метод, аналогичный используемым нами ранее в программируемом конвейере, см. листинг 13.4.

Листинг 13.4. Инициализация графики.

```
public bool InitializeGraphics()
{
    // Set our presentation parameters
    PresentParameters presentParams = new PresentParameters();
    presentParams.Windowed = true;
    presentParams.SwapEffect = SwapEffect.Discard;
    presentParams.AutoDepthStencilFormat = DepthFormat.D16;
    presentParams.EnableAutoDepthStencil = true;
    bool canDoHardwareSkinning = true;
    // Does a hardware device support shaders?
    Caps hardware = Manager.GetDeviceCaps(0, DeviceType.Hardware);
    // We will need at least four blend matrices
    if (hardware.MaxVertexBlendMatrices >= 4)
```

```
        // Default to software processing
        CreateFlags flags = CreateFlags.SoftwareVertexProcessing;
        // Use hardware if it's available
        if (hardware.DeviceCaps.SupportsHardwareTransformAndLight)
        flags = CreateFlags.HardwareVertexProcessing;
        // Use pure if it's available
        if (hardware.DeviceCaps.SupportsPureDevice)
        flags != CreateFlags.PureDevice;
        // Yes, Create our device
        device = new Device(0, DeviceType.Hardware, this, flags, presentParams);
    }
    else
    {
        //No shader support
        canDoHardwareSkinning = false;
        // Create a reference device
        device = new Device(0, DeviceType.Reference, this,
            CreateFlags.SoftwareVertexProcessing, presentParams);
    }
    // Create the animation
    CreateAnimation(@".,\..\tiny.x", presentParams);
    // Hook the device reset event
    device.DeviceReset += new EventHandler(OnDeviceReset);
    OnDeviceReset(device, null);
    return canDoHardwareSkinning;
}
```

Программа проверяет аппаратные возможности и при необходимости эмулирует программное устройство, поскольку файл анимации, используемый для данного примера, потребует по крайней мере четыре смешанные матрицы вершин.

Как только мы создали устройство, необходимо, определив соответствующий метод, загрузить анимацию. Перед добавлением метода загрузки анимации нам следует написать обработчик события сброса устройства для этого метода. Код обработчика событий, который необходимо добавить в наш класс, приведен в листинге 13.5.

Листинг 13.5. Обработчик события сброса устройства.

```
private void OnDeviceReset(object sender, EventArgs e)
{
    Device dev = (Device)sender;
    // Set the view matrix
    Vector3 vEye = new Vector3( 0, 0, -1.8f * objectRadius );
    Vector3 vUp = new Vector3( 0, 1, 0 );
```

```
    dev.Transform.View = Matrix.LookAtLH(vEye, objectCenter, vUp);
    // Setup the projection matrix
    float aspectRatio = (float)dev.PresentationParameters.BackBufferWidth
       / (float)dev.PresentationParameters.BackBufferHeight,
    dev.Transform.Projection = Matrix.PerspectiveFovLH( (float)Math.PI / 4,
       aspectRatio, objectRadius/64.0f, objectRadius*200.0f );
    // Initialize our light
    dev.Lights[0].Type = LightType.Directional;
    dev.Lights[0].Direction = new Vector3(0.0f, O.Of, l.Of);
    dev.Lights[0].Diffuse = Color.White;
    dev.Lights[0].Commit();
    dev.Lights[0].Enabled = true;
}
```

Как вы можете видеть, в программе устанавливается матрица вида;" камера размещается достаточно далеко, чтобы был виден весь объект, и направляется на центр объекта. Матрица проекции формируется с учетом достаточно большой задней плоскости, чтобы также вместить всю модель. Также имеется один направленный источник освещения.

Теперь все готово для создания анимации, добавьте следующий код:

```
private void CreateAnimation(string file, PresentParameters presentParams)
{
    // Create our allocate hierarchy derived class
    AllocateHierarchyDerived alloc = new AllocateHierarchyDerived(this);
    // Load our file
    rootFrame = Mesh.LoadHierarchyFromFile(file, MeshFlags.Managed,
       device, alloc, null);
    // Calculate the center and radius of a bounding sphere
    objectRadius = Frame.CalculateBoundingSphere(rootFrame.FrameHierarchy,
       out objectCenter);
    // Setup the matrices for animation
    SetupBoneMatrices((FrameDerived)rootFrame.FrameHierarchy);
    // Start the timer
    DXUtil.Timer(DirectXTimer.Start);
}
```

Метод кажется довольно простым, но в действительности он вызывает другие методы.

Сначала реализуется производный класс allocate hierarchy. Затем вызывается метод LoadffierarchyFromFile. При вызове этого метода следует учесть, что перегрузки CreateFrame и CreateMeshContainer вызываются неоднократно, в зависимости от числа фреймов и контейнеров MeshContainer в вашем объекте. Возвращаемый объект AnimationRootFrame будет включать корневой фрейм иерархичного дерева и контроллер анимации для этого объекта.

После создания иерархии рассчитывается граничная сфера всего фрейма (метод CalculateBoundingSphere). Метод CalculateBoundingSphere возвращает радиус этой сферы и центр объекта (который уже использовался для установки камеры).

И, последнее, создаются матрицы каркасов (Bone Matrices). Это первый метод, который будет обходить дерево иерархии фрейма, и он же будет базовым для остальных методов, приведенных в листинге 13.6.

Листинг 13.6. Создание матриц каркасов.

```
private void SetupBoneMatrices(FrameDerived frame)
{
   if (frame.MeshContainer != null)
   {
      SetupBoneMatrices((MeshContainerDerived)frame.MeshContainer);
   }
if (frame.FrameSibling != null)
{
   SetupBoneMatrices((FrameDerived)frame.FrameSibling);
}
if (frame.FrameFirstChild != null)
{
   SetupBoneMatrices((FrameDerived)frame.FrameFirstChild);
   }
}
private void SetupBoneMatrices(MeshContainerDerived mesh)
{
   // Is there skin information? If so, setup the matrices
   if (mesh.SkinInformation != null)
   {
      int numBones = mesh.SkinInformation.NumberBones;
      FrameDerived[] frameMatrices = new FrameDerived[numBones];
      for (int i = 0; i< numBones; i++)
      {
         FrameDerived frame = (FrameDerived)Frame.Find(
            rootFrame.FrameHierarchy,
            mesh.SkinInformation.GetBoneName(i));
         if (frame == null)
            throw new ArgumentException();
         frameMatrices[i] = frame;
      }
      mesh.SetFrames(frameMatrices);
   }
}
```

Как вы можете видеть, обход иерархии не представляется сложным. Если имеется родственный фрейм, вызывается непосредственно метод, в котором мы в настоящее время находимся. То же самое происходит, если имеются дочерние фреймы. Следует отметить, что при дочерних фреймах вызывать метод из самого себя целесообразно только один раз, поскольку последующие дочерние фреймы будут друг для друга родственными. В этом случае мы сохраняем каждый фрейм в соответствии с названием его каркаса.

Перегрузка, которая принимает фрейм, не представляет интереса, ее единственное назначение заключается в обходе иерархичного дерева и передаче контейнера MeshContainer во вторую перегрузку. Таким образом, мы создали массив фреймов (для каждого каркаса объекта), описывающий скелетную информацию, которая понадобится нам в дальнейшем.

ИСПОЛЬЗОВАНИЕ ТАЙМЕРА DIRECTX

Для системы анимации понадобится использование высокоточного таймера. Написанный нами код уже подразумевает использование таймера DirectX, который включен в SDK. Мы можем добавить этот файл к проекту, щелкнув по папке Add Existing Item и выбрав исходный файл утилиты dxutil.cs.

Теперь настало время переписать наш основной метод с учетом инициализации нашей новой графики, см. листинг 13.7.

Листинг 13.7. Основная точка входа.

```
static void Main()
{
   using (Form1 frm = new Form1())
   {
      // Show our form and initialize our graphics engine
      frm.Show();
      if (!frm.InitializeGraphics())
      {
         MessageBox.Show("Your card can not perform skeletal animation on " +
            "this file in hardware. This application will run in " +
            "reference mode instead.");
      }
      Application.Run(frm);
   }
}
```

Рендеринг анимированных объектов

Теперь нам осталось отобразить нашу анимацию на экране. Метод рендеринга сам по себе кажется обманчиво простым, см. добавленный к нашему классу код в листинге 13.8.

Листинг 13.8. Рендеринг анимированного бъекта.

```
protected override void OnPaint(System.Windows.Forms.PaintEventArgs e)
{
   ProcessNextFrame();
   device.Clear(ClearFlags.Target | ClearFlags.ZBuffer,
      Color.CornflowerBlue, 1.0f, 0);
   device.BeginScene(),
   // Draw our root frame
   DrawFrame((FrameDerived)rootFrame.FrameHierarchy);
   device.EndScene();
   device.Present();
   this.Invalidated();
}
```

Далее мы объявляем метод обработки фрейма «ProcessNextFrame», очищаем устройство и рисуем корневой фрейм:

```
private void ProcessNextFrame()
{
   // Get the current elapsed time
   elapsedTime = DXUtil.Timer(DirectXTimer.GetElapsedTime);
   // Set the world matrix
   Matrix worldMatrix = Matrix.Translation(objectCenter);
   device.Transform.World = worldMatrix;
   if (rootFrame.AnimationController != null)
      rootFrame.AnimationController.AdvanceTime(elapsedTime, null);
   UpdateFrameMatrices((FrameDerived)rootFrame.FrameHierarchy, worldMatrix);
}
```

Вначале сохраняется текущее время исполнения, параметр elapsed time. Затем создается мировая матрица для корневого фрейма (в принципе, это матрица перемещения в центр объекта, с последующим обновлением устройства). Полагая, что объект можно анимировать, мы устанавливаем временной параметр advance time, используя сохраненный параметр elapsed time. Теперь необходимо перезаписать матрицы преобразования:

```
private void UpdateFrameMatrices(FrameDerived frame, Matrix parentMatrix)
{
   frame.CombinedTransformationMatrix = frame.TransformationMatrix *
      parentMatrix;
   if (frame.FrameSibling != null)
   {
      UpdateFrameMatrices((FrameDerived)frame.FrameSibling, parentMatrix);
   }
   if (frame.FrameFirstChild != null)
   {
      UpdateFrameMatrices((FrameDerived)frame.FrameFirstChild,
         frame.CombinedTransformationMatrix);
   }
}
```

В этом методе общая матрица преобразования текущего фрейма получена путем умножения матрицы преобразования имеющегося фрейма на родительскую матрицу преобразования. Каждый из родственных фреймов использует соответствующую родительскую матрицу текущего фрейма. Каждый из дочерних фреймов должен использовать общую матрицу преобразования текущих фреймов, чтобы объединить ее с собственной. Таким образом, формируется цепочка матриц, где окончательный дочерний фрейм имеет собственную матрицу преобразования, объединенную с каждой родительской матрицей.

Теперь мы можем нарисовать наш объект, используя знакомый нам метод рисования DrawFrame:

```
private void DrawFrame(FrameDerived frame)
{
   MeshContainerDerived mesh = (MeshContainerDerived)frame.MeshContainer;
   while(mesh != null)
   {
      DrawMeshContainer(mesh, frame);
      mesh = (MeshContainerDerived)mesh.NextContainer;
   }
   if (frame.FrameSibling != null)
   {
      DrawFrame((FrameDerived)frame.FrameSibling);
   }
   if (frame.FrameFirstChild != null)
   {
      DrawFrame((FrameDerived)frame.FrameFirstChild);
   }
}
```

Здесь выполняется знакомый нам обход иерархии, программа пытается нарисовать каждый контейнер MeshContainer, на который ссылается используемый фрейм. Для рендеринга соответствующего MeshContainer добавьте к приложению метод, приведенный в листинге 13.9.

Листинг 13.9. Рендеринг контейнера MeshContainer.

```
private void DrawMeshContainer(MeshContainerDerived mesh, FrameDerived
frame)
 {
    // Is there skin information?
    if (mesh.SkinInformation != null)
     {
        int attribIdPrev = -1;
        // Draw
        for (int iattrib = 0; iattrib < mesh.NumberAttributes; iattrib++)
        {
           int numBlend = 0;
           BoneCombination[] bones = mesh.GetBones() ;
           for (int i = 0; i < mesh.NumberInfluences; i++)
            {
               if (bones[iattrib].BoneId[i] != -1)
                {
                   numBlend = i;
                }
        }
        if (device.DeviceCaps.MaxVertexBlendMatrices >= numBlend + 1)
        {
           // first calculate the world matrices for the current set of

           // blend weights and get the accurate count of the number of
           // blends
           Matrix[] offsetMatrices = mesh.GetOffsetMatrices();
           FrameDerived[] frameMatrices = mesh.GetFrames();
           for (int i = 0; i < mesh.NumberInfluences; i++)
            {
               int matrixIndex = bones[iattrib].BoneId[i];
               if (matrixIndex != -1)
               {
                  Matrix tempMatrix = offsetMatrices[matrixIndex] *
                     frameMatrices[matrixIndex].
                     CombinedTransformationMatrix;
                  device.Transform.SetWorldMatrixByIndex(i, tempMatrix);
               }
            }
```

```
            device.RenderState.VertexBlend = (VertexBlend)numBlend;
            // lookup the material used for this subset of faces
            if ((attribldPrev != bones[iattrib].Attribld) ||
             (attribldPrev == -1))
             {
                device.Material = mesh.GetMaterials() [
                   bones[iattrib].Attribld].Material3D;
                device.SetTexture(0, mesh.GetTextures() [
                   bones[iattrib].Attribld]) ;
                attribldPrev = bones[iattrib].Attribld;
             }
            mesh.MeshData.Mesh.DrawSubset(iattrib);
         }
      }
   )
   else // standard mesh, just draw it after setting material properties
   {
      device.Transform.World = frame.CombinedTransformationMatrix;
         ExtendedMaterial[] mtrl = mesh.GetMaterials();
         for (int iMaterial = 0; iMaterial < mtrl.Length; iMaterial++)
         {
            device.Material = mtrl[iMaterial].Material3D;
            device.SetTexture(0, mesh.GetTextures() [iMaterial]);
            mesh.MeshData.Mesh.DrawSubset(iMaterial);
         }
      }
   }
```

Размер листинга данного метода впечатляет, хотя, если рассматривать его по частям, это выглядит не так сложно. Вначале проверяются данные об оболочке, значение SkinInformation. Если контейнер Meshcontainer не содержит никакой скелетной информации, объект будет рендирован точно так же, как мы делали это ранее при отображении обычных объектов. При наличии скелетной информации способы рендеринга могут быть различными.

АНИМАЦИЯ ОБЪЕКТОВ, НЕ ИМЕЮЩИХ КАРКАСА

Если объект не содержит никакой скелетной информации, это не означает, что объект не может быть анимирован. Если в качестве анимации, включенной в объект, используется стандартная матричная операция (например: масштабирование, перемещение или вращение), нет никакой необходимости в каркасах и скелетных данных. Однако система анимации будет переписывать матрицы для вашего объекта в соответствии с обычными методами.

Далее в листинге для каждого принимаемого атрибута объекта (набор материалов, текстур и пр.) выполняются различные операции. Вначале просматривается таблица комбинаций каркасов, и определяются весовые коэффициенты, которые объект будет использовать. Файл примера, включенный к CD диск, использует максимум четыре весовых параметра, при этом подразумевается, что устройство имеет возможность объединять и оперировать большим количеством матриц.

После проверки того, что устройство может отображать объект с определенными весовыми коэффициентами, необходимо выполнить преобразование координат.

Для каждого элемента, найденного по идентификационному номеру в таблице комбинаций каркасов, матрица смешения объединяется с общей матрицей преобразования фрейма, и текущая индексированная мировая матрица, преобразованная в результирующую матрицу, сохраняется. Это позволит приложению Direct3D отобразить каждую интерполированную вершину (blended vertex) в соответствии с преобразованием координат.

После выполнения этих операций устанавливается состояние рейдера интерполированной вершины в соответствии с номером numBlend объекта. И, наконец, устанавливаются и затем отображаются материал и текстура данного подмножества.

Запуская приложение, мы должны увидеть на экране модель, идущую по направлению к нам, **рис.13.**1

Рис. 13.1. Аиимированный объект

ИСПОЛЬЗОВАНИЕ ОБЪЕКТОВ «INDEXED MESH» ДЛЯ АНИМАЦИИ

Ранее мы обсуждали использование для рендеринга вершин индексных буферов, что позволяло сократить число необходимых для рисования полигонов и более рационально использовать память при отображении. При визуализации сложных объектов, таких как в нашем примере, преимущества использования индексированных объектов налицо. Кроме того, это позволяет упростить написание кода.

Прежде чем модифицировать наш объект, необходимо выполнить некоторые действия. Сначала следует добавить новый объект к производному классу mesh container:

```
private int numPal = 0;
public int NumberPaletteEntries
{
   get { return numPal; } set { numPal = value; }
}
```

Это позволит сохранить число каркасных матриц, которые могут использоваться для матричной палитры skinning при преобразовании нашего объекта. Затем, поскольку мы больше не будем использовать стандартную интерполяцию вершин для нашей анимации, необходимо переписать метод инициализации, проверяя возможность поддержки устройством новых операций. Для этого перепишите строку, включающую данную проверку:

```
if (hardware.MaxVertexBlendMatrixIndex >= 12)
```

Все, что теперь требуется, — заменить запрос создания объекта (листинг 13.10) и затем вызов рисунка (листинг 13.11).

Листинг 13.10. Создание объекта Mesh.

```
public void GenerateSkinnedMesh(MeshContainerDerived mesh)
 {
   if (mesh.SkinInformation == null)
      throw new ArgumentException();
   int numMaxFaceInfl;
   MeshFlags flags = MeshFlags.OptimizeVertexCache;
   MeshData m = mesh.MeshData;
   using(IndexBuffer ib = m.Mesh.IndexBuffer)
    {
      numMaxFaceInfl = mesh.SkinInformation.GetMaxFaceInfluences(ib,
         m.Mesh.NumberFaces);
```

```
}
//12 entry palette guarantees that any triangle (4 independent
// influences per vertex of a tri) can be handled
numMaxFacelnfl = (int)Math.Min(numMaxFacelnfl, 12);
if (device.DeviceCaps.MaxVertexBlendMatrixIndex + 1 >= numMaxFacelnfl)
{
   mesh.NumberPaletteEntries = (int)Math.Minf(device.DeviceCaps.
      MaxVertexBlendMatrixIndext 1) / 2 ,
      mesh.SkinInformation.NumberBones);
   flags != MeshFlags.Managed;
}
BoneCombination[] bones;
int numlnfl;
m.Mesh = mesh.SkinInformation.ConvertToIndexedBlendedMesh(m.Mesh, flags,
   mesh.GetAdjacencyStreamf), mesh.NumberPaletteEntries, out numlnfl,
out bones);
mesh.SetBones(bones);
mesh.Numberlnfluences = numlnfl;
mesh.NumberAttributes = bones.Length;
mesh.MeshData = m;
```

Вначале определяется максимальное число влияний поверхности (face influences), в этом объекте их необходимо по крайней мере 12 (4 значения интерполяции для каждой вершины в треугольнике). Полагая, что наше устройство поддерживает эту возможность (проверяется в методе инициализации), мы вычисляем число входов палитры (либо число каркасов, либо половина от максимального числа поддерживаемых влияний поверхности).

Теперь мы можем преобразовывать наш объект в индексный и сохранять те же самые данные, которые мы использовали в нашей неиндексированной версии. Итак, заменяем вызов рисунка, см. листинг 13.11. Для краткости мы включаем только код внутри блока, подразумевая, что значение оболочки «skin» ненулевое.

Листинг. 13.11. Вызов рисунка.

```
if (mesh.Numberlnfluences == 1)
   device.RenderState.VertexBlend = VertexBlend.ZeroWeights;
else
   device.RenderState.VertexBlend = (VertexBlend)(mesh.Numberlnfluences - 1);
if (mesh.Numberlnfluences > 0)
   device.RenderState.IndexedVertexBlendEnable = true;
BoneCombination[] bones = mesh.GetBonesf);
for(int iAttrib = 0; iAttrib < mesh.NumberAttributes; iAttrib++)
{
```

```
    // first, get world matrices
    for (int iPaletteEntry = 0; iPaletteEntry < mesh.NumberPaletteEntries;
        ++iPaletteEntry)
    {
        int iMatrixIndex = bones[iAttrib].BoneId[iPaletteEntry];
        if (iMatrixIndex != -1)
        {
            device.Transform.SetWorldMatrixByIndex(iPaletteEntry,
            mesh.GetOffsetMatricesO [iMatrixIndex] *
            mesh.GetFramesO [iMatrixIndex].
            CombinedTransformationMatrix);
        }
    }
    // Setup the material
    device.Material = mesh.GetMaterials()[bones[iAttrib].Attribld].Material3D; '
    device.SetTexture(0, mesh.GetTextures()[bones[iAttrib].Attribld]) ;
    // Finally draw the subset
    mesh.MeshData.Mesh.DrawSubset(iAttrib);
}
```

Этот метод более прост в написании. Вначале мы определяем состояние рендера интерполированной вершины как «-1». Затем, если есть влияния (для случая скелетной анимации), состояние рендера принимает значение «true» для параметра IndexedVertexBlendEnable.

Остальная часть кода подобна предыдущему методу. Для каждого входа палитры записывается соответствующая мировая матрица с учетом индекса общей матрицы смещения и фреймов общей матрицы преобразования. После записи мировых матриц можно устанавливать материалы, текстуры и рисовать каждое подмножество. Использование индексированного mesh-объекта более предпочтительно в плане быстродействия (от 30% и более, в зависимости от типа данных), чем использование обычного mesh-объекта.

Краткие выводы

В этой главе мы ознакомились с основами анимации, включая скелетную анимацию. Основные рассмотренные в этой главе темы.

- Создание и загрузка иерархической системы фреймов.
- Просмотр иерархии, рендеринг фреймов.
- Формирование скелетных данных.
- Использование контроллера анимации.
- Использование «indexed mesh».

В следующей главе мы обсудим использование звука в создаваемых приложениях.

ЧАСТЬ IV

ЗВУК И УСТРОЙСТВА ВВОДА

Глава 14. Добавление звука

Глава 15. Управление устройствами ввода

Глава 14. Добавление звука

Ни одно графическое приложение или игра не будет живым, если вы не добавили соответствующее окружение, атмосферу и особенно звук. Можете ли вы представить известную игру Pac-Man без знакомого звукового сопровождения? В этой главе мы изучим добавление звука в наши приложения. Темами, которые мы обсудим, являются.

- Загрузка и проигрывание статических звуков.
- Проигрывание звуков в 3D пространстве.
- Проигрывание звуков с эффектами.

Включение пространства имен SOUND

Звуковые ресурсы, которые мы будем использовать, отсутствуют в списке имен Direct3D, вместо этого необходимо использовать пространство имен Microsoft.DirectX.DirectSound. Для каждого из примеров, которые будут рассмотрены в этой главе, необходимо добавить ссылку на указанную сборку, также как и директиву using.

Загрузка и проигрывание статических звуков

В начале выясним, что мы хотим сделать для добавления звука? Например, иметь некоторый тип звуковых данных, выводимых на динамики, приложенные к системе. Подобно классу Device для приложения Direct3D, который управляет аппаратным обеспечением компьютерной графики, в приложении DirectSound имеется класс Device, который управляет звуковыми аппаратными средствами. Поскольку оба этих классов совместно используют одно и тоже название (в различных пространствах имен), необходимо строго описывать ссылки на переменную Device, если мы подразумеваем использование Direct3D и DirectSound в одном файле кода.

Подобно графическим буферам в Direct3D, приложение DirectSound имеет буферы для обработки, передачи и вывода звуковых данных. В DirectSound существуют только два типа буферов: либо основной класс Buffer, либо производный от него объект SecondaryBuffer.

Итак, вначале мы создаем новый проект, включаем в него ссылки на DirectSound, а затем добавляем следующие объявления переменных:

```
private Device device = null;
private SecondaryBuffer sound = null;
```

Для проигрывания загружаемых звуковых данных нам нужно иметь одно звуковое устройство и один буфер SecondaryBuffer. Теперь мы можем записать метод инициализации звука InitializeSound, который позволяет запус-

кать и проигрывать звук. Поскольку указанный метод будет ссьшаться на звуковой файл, необходимо это предусмотреть. Используемый в этом методе звуковой файл скопирован из DirectX SDK. Добавьте следующий код:

```
public void InitializeSound()
{
    device = new Device ();
    device.SetCooperativeLevel(this, CooperativeLevel.Normal);
    sound = new SecondaryBuffer(@"..\..\drumpad-crash.wav", device);
    sound.Play(0, BufferPlayFlags.Default);
}
```

Приведенный здесь код достаточно прост. Используя простой конструктор параметров, мы создаем новое устройство (другие варианты загрузки этого объекта мы рассмотрим позднее) и определяем уровень совместного доступа к устройству CooperativeLevel.Normal. Поясним подробнее.

Звуковые устройства компьютера используются многими приложениям. Система Windows может подавать звуковой сигнал каждый раз, когда возникает ошибка. Система Messenger может подавать звуковой сигнал, когда кто-то вошел или зарегистрировался, и все это может происходить, например, во время прослушивания музыки. Уровень совместного доступа используется, чтобы определить степень совместного с другими приложениями использования звуковых карт. В нашем примере данный параметр определен по умолчанию. Это подразумевает многозадачный режим и совместно используемые ресурсы. Описание других уровней доступа, например, приоритетных, можно найти в документации DirectX SDK.

После установки уровня доступа можно создать или загрузить буфер из файла (код, включенный в CD диск DirectX SDK, использует простой звук) и затем уже проигрывать звук. Первый параметр вызова проигрывания звука — приоритет, который имеет смысл только, если буфер создается с задержкой (чего мы пока не имеем). Теперь, используя заданные по умолчанию флажки, вызываем процедуру проигрывания содержимого буфера:

```
static void Main()
{
    using (Forml frm = new Forml())
    {
        frm.InitializeSoundO ;
        Application.Run(frm);
    }
}
```

Все достаточно просто. Несколько строк программы, и при запуске приложения появляется звук.

Конструктор для буфера SecondaryBuffer требует несколько отличных параметров. Рассмотрим две основных разновидности этого конструктора:

public SecondaryBuffer (System.String *fileName* ,
Microsoft.DirectX.DirectSound.BufferDescription *desc* ,
Microsoft.DirectX.DirectSound.Device parent)

public SecondaryBuffer (System.10.Stream *source* , System.Int32 length ,
Microsoft.DirectX.DirectSound.BufferDescription *desc* ,
Microsoft.DirectX.DirectSound.Device parent)

Все конструкторы вторичного буфера оперируют с устройством, которое будет использоваться для проигрывания звуковых данных. Конструкторы могут принимать либо *имя JileName* файла загружаемых звуковых данных, либо поток, который содержит звуковые данные. Для описания различных опций буфера, который вы создаете, существует конструктор BufferDescription. Все создаваемые буферы имеют характеристики, и, если вы не задаете конструктор BufferDescription, то значения его параметров и флажков устанавливаются по умолчанию. Свойства конструктора BufferDescription приведены в таблице 14.1.

Таблица 14.1. Свойства и флажки конструктора **BufferDescription**

Название	Описание
BufferBytes	Размер буфера в байтах (чтение-запись). Если вы создаете первичный буфер или буфер из потока или звукового файла, вы можете оставить этотму параметру нулевое значение
CanGetCurrentPosition	Атрибут чтение-запись. Является булевой переменной, уточняет, нужна ли точная позиция проигрывания
Contro3D	Атрибут чтение-запись. Является булевой переменной, сообщает, можно ли манипулировать вашим буфером в 3D-пространстве. Эта опция не может использоваться, если ваш буфер содержит формат стерео (два канала), или опция ControlPan установлена в значение «true»
ControlEffects	Атрибут чтение-запись. Является булевой переменной, указывает на то, что буфер может или не может использовать обработку эффектов. Чтобы использовать эти эффекты, необходимо иметь 8- или 16-разрядные аудио данные формата PCM, при этом возможно использование не более двух каналов (стерео)

Название	Описание
ControlFrequency	Атрибут чтение-запись. Является булевой переменной, устанавливает возможность изменения частоты обращения к буферу (при значении «true»)
ControlPan	Атрибут чтение-запись. Является булевой переменной, определяет возможность поддержки опции panning (панорама), не может использоваться с флажком Contro3D
ControlPositionNotify	Атрибут чтение-запись. Является булевой переменной, показывает, поддерживает ли буфер указание на позицию
ControlVolume	Атрибут чтение-запись. Является булевой переменной, указывает на то, что буфер может или не может управлять громкостью
DeferLocation	Атрибут чтение-запись. Является булевой переменной, указывает, может ли буфер присоединяться к устройству во время проигрывания. Данный флажок должен принимать значение «true» для буферов, поддерживающих голосовое управление.

Flags	Поразрядная комбинация перечисления BufferDescriptionFlags. Значения для этих элементов установлены как логические переменные, например:
	
	desc. ControlPan = true;
	desc.ControlEffects = true;
	
	что аналогично следующему:
	
	desc.Flags = BufferDescriptionFlags.ControlPan BufferDescriptionFlags.ControlEffects;

Format	Атрибут чтение-запись. Определяет акустический формат создаваемых или загружаемых в буфер звуковых данных
GlobalFocus	Атрибут чтение-запись. Булева переменная. При значении по умолчанию («false») звук проигрывается, если приложение активно. При значении «true» звук будет проигрываться в любом состоянии приложения
Guid3DAlgorithm	Атрибут чтение-запись. Идентификатор GUID. Определяет алгоритм, используемый для 3D-виртуализации. Вы можете использовать любую из констант DSoundHelper.Guid3Dxxxx, перечисленных в Управляемом DirectX для встроенных режимов

Название	Описание
LocatelnHardware	Атрибут чтение-запись, булева переменная. Определяет обработку буфера аппаратными средствами. Если значением является «true», а необходимая аппаратная поддержка отсутствует, при создании буфера произойдет ошибка
LocatelnSoftware	Атрибут чтение-запись, булева переменная. Определяет обработку буфера программными средствами, значение «true» разрешает программную обработку независимо от аппаратных ресурсов
Mute3DAtMaximumDistance	Атрибут чтение-запись, булева переменная, определяет остановку проигрывания при достижении максимальной дистанции. Параметр применим только к программным буферам
PrimaryBuffer	Атрибут чтение-запись. Показывает, является ли буфер первичным.
StaticBuffer	Атрибут чтение-запись, булева переменная. Определяет размещение буфера в аппаратной памяти (значение «true»), если она поддерживается, либо в программной памяти, если аппаратная недоступна. При использовании данного флажка вы не можете задавать значение «true» для флажка ControlEffects (см. выше)
StickyFocus	Атрибут чтение-запись. Булева переменная. Определяет использование закрепленного фокуса. Как уже упоминалось при обсуждении флажка глобального фокуса, значение по умолчанию для DirectSound прекращает проигрывание буфера, если ваше окно не активно. Если флажок установлен в значение «true», буфер будет продолжать проигрывание, даже если окно уже не использует DirectSound

Таким образом, имеется несколько опций для буфера. В следующем примере мы рассмотрим работу некоторых из них. Мы постараемся управлять некоторыми параметрами, как-то: перетекание звука, панорама звука и т. д.

Чтобы выполнить это, нам необходимо использовать определение класса «buffer description». Добавьте следующий код после вызова установки уровня доступа SetCooperativeLevel:

```
BufferDescription desc = new BufferDescription(
desc.ControlPan = true;
desc.GlobalFocus = true;
```

По существу, мы сообщаем приложению DirectSound, что создадим буфер и будем управлять панорамой звука. Также определяем буфер как глобальную переменную.

Теперь мы должны определить процедуру создания буфера с помощью конструктора, использующего описание уже созданного нами буфера. Для улучшения эффекта мы зацикливаем буфер на время проигрывания. Для выполнения этого измените код следующим образом:

```
sound = new SecondaryBuffer(@".,\..\drumpad-crash.wav",desc, device);
sound.Play(0, BufferPlayFlags.Looping);
```

При запуске приложения произойдет непрерывное проигрывание звукового файла, но при этом мы пока еще не изменяли панорамное значение нашего буфера. Для разнообразия можно обновлять его каждые 50 миллисекунд. Для в этого в окне design view нашей формы добавим таймер и установим интервал 50 миллисекунд. Добавьте следующий код обработчика таймера (либо дважды кликните на таймер в окне design view):

```
private void timerl_Tick(object sender, System.EventArgs e)
{
   // Adjust the pan
   sound.Pan *= -1;
}
```

Панорамное значение — целочисленное значение, которое определяет позицию стерео звука; чем «отрицательнее» значение, тем больше звука выходит из левого динамика. Чем больше положительное значение — из правого. Нулевое значение (значение по умолчанию) уравнивает звук из обоих динамиков. Теперь установите начальное значение панорамирования на левый динамик и запустите таймер. Добавьте этот код непосредственно после вызова проигрывания в InitializeSound:

```
sound.Pan = -5000;
timerl.Enabled = true;
```

Добавив таймер, запускаем приложение. Мы должны услышать проигрывание нашего звукового файла, при этом звук должен перетекать из динамика в динамик. Как оказалось, использовать звуки для нашего приложения — не такая сложная процедура.

Использование 3D-звука

Многие компьютерные мультимедийные системы в настоящее время используют новейшие модели колонок и звуковых плат класса «Hi-End».

Уже трудно кого-нибудь удивить системами объемного звучания, а для разработчиков компьютерных игр это вообще является нормой.

Приложение DirectSound API уже имеет возможность запускать звуки в трехмерном пространстве и может использовать их особенности (если компьютер поддерживает эти опции). Опции управляются посредством объектов Buffer3D (чтобы управлять источником звука) и Listener3D (чтобы управлять позицией и ориентацией слушателя). Обратите внимание на то, что каждый из этих конструкторов использует 3D-звук. Эти буферы должны создаваться с разрешенным флажком ControBD (значение «true»).

Какие же преимущества дает 3D-звук? Это наиболее ощутимо, когда мы находимся в кинотеатре, где используются системы объемного звучания, когда различные объекты издают звук из различных динамиков. Современные игры также требуют все большего развития систем воспроизведения звука.

Когда мы хотим манипулировать 3D-звуком, мы можем управлять источниками звука или «ушами» (то есть слушателем), принимающими звук. Для начала, используя объект Buffer3D, мы можем рассмотреть перемещение вокруг источника звука

Этот объект-класс предназначен для управления 3D-настройками буфера. Параметры объекта Buffer3D и их описание приведены в таблице 14.2.

Таблица 14.2. Параметры и свойства объекта Buffer3D

Название	Описание
ConeAngles	Атрибут чтение-запись. Используется для установки углов относительно конуса проекции. Оба угла (внутренний и внешний) определены в градусах. Звуки в пределах внутреннего конического угла находятся на нормальном уровне, в то время, как вне его — на окружающем фоновом уровне
ConeOrientation	Атрибут чтение-запись. Используется для ориентирования конуса проекции. Приложение DirectSound автоматически нормализует соответствующий вектор (центральный вектор конуса проекции)
ConeOutsideVolume	Атрибут чтение-запись. Используется для регулировки уровня звука, находящегося за пределами угла звукового конуса
Deferred	Атрибут чтение-запись. Булева переменная. Определяет, изменяются ли регулируемые параметры сразу либо они задерживаются, пока пользователь изменяет установки. По умолчанию данное значение «false», то есть свойства изменяются сразу

Название	Описание
MaxDistance	Атрибут чтение-запись. Определяет максимальное расстояние до слушателя, при котором звук еще не затухает
MinDistance	Атрибут чтение-запись. Определяет минимальное расстояние до слушателя, при котором звук начинает затухать
Mode	Атрибут чтение-запись. Определяет режим звуковой обработки. Значение по умолчанию — режим Mode3D.Normal. Вы можете также использовать Mode3D.Disable, который отключает обработку 3D-звука, или Mode3D.HeadRelative, который подразумевает регулировку звука слушателем
Position	Атрибут чтение-запись. Определяет текущее местоположение источника звука в мировых координатах
Velocity	Атрибут чтение-запись. Определяет скорость перемещения источника звука (по умолчанию значение в метрах на секунду)

Имеется и другой флажок AllParameters, позволяющий изменять все установки сразу. Можно попробовать переписать последний пример, использующий параметр панорамы, установив вместо него параметр 3Dprocessing.

Звуковой файл, который мы использовали до настоящего времени, имеет стерео звучание, т. е. имеет два отдельных канала для левого и правого динамиков. Чтобы использовать трехмерную обработку, мы должны использовать файл с монозвуком (или стерео, но с только одним каналом). Таким образом, для нашего примера будем загружать монозвуковой файл с CD диска DirectX SDK. Добавьте ссылку на объект Buffer3D, который будет использоваться при управлении трехмерным буфером:

```
private Buffer3D buffer = null;
```

Также необходимо переписать метод инициализации InitializeSound, чтобы использовать 3D-обработку вместо панорамирования, которое мы использовали прежде:

```
public void InitializeSound!)
(
    device = new Device();
    device.SetCooperativeLevel(this,    CooperativeLevel.Normal);
    BufferDescription desc = new BufferDescription();
```

```
    desc.Control3D = true;
    desc.GlobalFocus = true;
    sound = new SecondaryBuffer(@".A..\drumpad-bass drum.wav", desc, device);
    buffer = new Buffer3D(sound);
    sound.Play(0, BufferPlayFlags.Looping);
    buffer.Position = new Vector3(-0.1f, 0.0f, 0.0f);
    timer1.Enabled = true;
}
```

Мы заменили панорамирование трехмерной обработкой, плюс, заменили звуковой файл. Затем создали объект Buffer3D, установив его на непрерывное проигрывание (значение loop). Далее устанавливается проигрывание по левому каналу (напомним, что по умолчанию это значение 0, 0, 0).

Теперь необходимо переписать код таймера и определить круговое звучание:

```
private void timer1_Tick(object sender, System.EventArgs e)
{
    // Adjust the position
    buffer.Position *= mover;
    if ((Math.Abs(buffer.Position.X) > MoverMax) && (mover == MoverUp))
    {
        mover = MoverDown;
    }
    if ((Math.Abs(buffer.Position.X) < MoverMin) && (mover == MoverDown))
    {
        mover = MoverUp;
    }
)
```

Как мы видим, в этом коде появились несколько новых переменных и констант, определяющих круговое звучание.

Для правильной компиляции их необходимо добавить к нашему классу:

```
private const float MoverMax = 35.0f;
private const float MoverMin = 0.5f;
private const float MoverUp = -1.05f;
private const float MoverDown = -0.95f;
private float mover = MoverUp;
```

Запуская приложение, обратите внимание на то, как звук перемещается от динамика к динамику, а также медленно удаляется, пропадает и возвращается вновь. Приложение будет выполняться, пока вы не выйдете из него.

Управление слушателем

Объект Listener3D создается и используется так же, как объект Buffer3D. Однако, вместо управления фактическим звуковым источником (которых может быть несколько), мы имеем дело непосредственно со слушателем, который может быть в устройстве только одним. По этой причине, при создании объекта listener мы не можем использовать SecondaryBuffer, а только первичный объект Buffer.

Прежде чем написать код, используя объект listener, необходимо учесть свойства и параметры настройки, которые мы можем изменять (аналогично тому, что мы делали с объектом 3D Buffer). Параметры и свойства приведены в таблице 14.3.

Таблица 14.3. Параметры и свойства объекта Listener3D

Название	Описание
CommitDeferredSettings	При установленной задержке параметров настройки в 3D буферах или листенере этот метод перешлет все новые значения в то же самое время. Метод не выполняется при отсутствии задержки установленных изменений
DistanceFactor	Атрибут чтение-запись. Устанавливает число метров в векторном модуле
DopplerFactor	Атрибут чтение-запись. Определяет коэффициент для эффекта Доплера. Любой буфер, который работает со скоростью перемещения источника звука, учитывает изменения тона, связанные с эффектом Доплера
Deferred	Определяет то, изменяются ли регулируемые параметры сразу либо насколько они задерживаются, пока пользователь изменяет установки. По умолчанию данное значение «false», то есть свойства изменяются сразу
Orientation	Атрибут чтение-запись. Определяет местоположение слушателя. Приложение DirectSound автоматически нормализует соответствующий вектор
RolloffFactor	Атрибут чтение-запись. Определяет параметр rolloff factor — скорость затухания по мере увеличения расстояния
Position	Атрибут чтение-запись. Определяет текущее местоположение слушателя в мировых координатах
Velocity	Атрибут чтение-запись. Определяет скорость перемещения слушателя (по умолчанию значение в метрах на секунду)

Теперь мы должны взять имеющийся пример, в котором перемещается звуковой буфер, и переписать код, вставив в него перемещение слушателя. Можно убрать ссылку на объект 3D buffer, который мы использовали, и заменить соответствующую переменную нижеприведенными:

```
private Listener3D listener = null;
private Microsoft.DirectX.DirectSound.Buffer primary = null;
```

Мы должны использовать первичный буфер для создания объекта listener. Также необходимо учесть, что поскольку имя класса Buffer находится в пространстве имен системы, мы должны полностью определить переменную buffer.

Для этого нужно переписать метод инициализации InitializeSound. Замените код создания звукового буфера на следующий:

```
BufferDescription primaryBufferDesc = new BufferDescription0;
primaryBufferDesc.Control3D = true;
primaryBufferDesc.PrimaryBuffer = true;
primary = new Microsoft.DirectX.DirectSound.Buffer(primaryBufferDesc,
device);
listener = new Listener3D(primary);
listener.Position = new Vector3(0.1f, 0.0f, 0.0f);
```

Здесь мы создаем первичный буфер, объявляем объект listener и затем устанавливаем его местоположение (несколько «левее» от заданного по умолчанию). Теперь мы должны переписать код таймера, чтобы перемещать слушателя:

```
private void timerl_Tick(object sender, System.EventArgs e)
{
   // Adjust the position
   listener.Position *= mover;
   if ((Math.Abs(listener.Position.X) > MoverMax) && (mover == MoverUp))

      mover = MoverDown;

   if ((Math.Abs(listener.Position.X) < MoverMin) && (mover == MoverDown))

      mover = MoverUp;

}
```

Легко заметить, что мы не сделали ничего, кроме замены буфера на слушателя. Этот пример должен проигрываться аналогично предыдущему.

ФЛАЖОК CONTROL3D

Важно обратить внимание, что для создания объекта listener используется флажок Control3D, в противном случае звук будет проигрываться с одинаковым уровнем громкости для всех динамиков.

Использование звуковых эффектов

Кроме обычных 3D-манипуляций со звуком, мы в нашем приложении можем использовать различные эффекты. Например, звук шагов по деревянным полам в маленькой комнате отличался бы от звука шагов, раздающихся в большом концертном зале. Крики в пещере вызвали бы многократное эхо, отражаемое от стен пещеры. Эти эффекты — неотъемлемая часть того реального погружения, которое вы будете пробовать создавать при разработке игры.

БУФЕРЫ, ПОДДЕРЖИВАЮЩИЕ ЗВУКОВЫЕ ЭФФЕКТЫ

Звуковые эффекты и методы их программирования могут быть реализованы только с использованием вторичного буфера SecondaryBuffer и соответствующих ссылок на этот объект.

К счастью, действие звуковых эффектов в достаточной мере описано и классифицировано, и несколько таких эффектов уже встроены в DirectSound API.

Перепишем наш старый пример, используя некоторые звуковые эффекты.

Во-первых, удалите таймер, так как здесь он не будет использоваться. Затем перепишите метод инициализации InitializeSound следующим образом:

```
public void InitializeSound()
{
   device = new Device));
   device.SetCooperativeLevel(this, CooperativeLevel.Normal);
   BufferDescription desc = new BufferDescription() ;
   desc.ControlEffects = true;
   desc.GlobalFocus = true;
   sound = new SecondaryBuffer(@"..\..\drumpad-crash.wav",desc, device);
   EffectDescription[] effects = new EffectDescription[1];
   effects[0].GuidEffectClass = DSoundHelper.StandardEchoGuid;
   sound.SetEffects(effects);
   sound.Play(0, BufferPlayFlags.Looping) ;
}
```

Самое большое изменение здесь — замена управления панорамой звука на управление звуковыми эффектами. Создается также массив описаний эффекта (в настоящем примере с только одним членом), и задается стандартный эффект — эхо. Затем созданные эффекты пересылаются, и запускается проигрывание буфера.

УСТАНОВКА ЭФФЕКТОВ

Эффекты могут быть установлены только на остановленном буфере. Если буфер проигрывается, вызов SetEffects игнорируется. Вы можете просмотреть свойство состояния буфера (Status property, статус буфера), чтобы определить, проигрывается буфер или нет. Также вы можете вызывать оператор Stop перед обращением к эффектам.

Определив и установив эффекты, мы можем запустить приложение. При проигрывании мы должны услышать эффект эха, которого не было до сих пор. Возможно и добавление «пачки» наложенных друг на друга эффектов (stacks). Чтобы использовать эффект эха вместе с эффектом гребня, сопровождаемым эффектом искажения, мы можем переписать метод следующим образом:

```
EffectDescription[] effects = new EffectDescription[3];
effects[0].GuidEffectClass = DSoundHelper.StandardEchoGuid;
effects[1].GuidEffectClass = DSoundHelper.StandardFlangerGuid;
effects[2].GuidEffectClass = DSoundHelper.StandardDistortionGuid;
sound.SetEffects(effects);
```

Мы получили некоторую смесь звуков барабана кимвал и работающего двигателя самолета (этот звук очень понравился автору книги — прим. ред.). Как уже упоминалось, существует несколько встроенных типов эффектов, см. таблицу 14.4.

Таблица 14.4. Типы встроенных звуковых эффектов

Название	Описание
Chorus	Эффект хора, удваивает голоса, выводя первоначальный звук второй раз с небольшой задержкой и слегка модулируя эту задержку
Compression	Эффект сжатия — по существу уменьшает сигнал до некоторой амплитуды.

Название	Описание
Distortion	Эффект искажения. Осуществляется добавлением гармоник к сигналу, при увеличении уровня изменяется форма волны
Echo	Эффект эхо, повторяет многократно первоначальный звук с заданной задержкой и спадом амплитуды последующего сигнала
Environment Reverberation.	Эффект отражения или раскаты. Осуществляется спецификацией 3D Audio Level 2 (I3DL2)
Flange	Эффект гребня, напоминает эффект хора, плюс эффект эхо только с маленькой задержкой и сменой тона через какое то время
Gargle	Просто модулирует амплитуду сигнала
Parametric Equalizer	Этот эффект действует как эквалайзер, позволяя усиливать или подавлять сигналы определенной частоты
Waves Reverberation	Этот эффект предназначен для использования вместе с музыкой, напоминает эффект раскатов

Теперь мы можем выбрать и применить любой из эффектов, выполнив предварительно соответствующие настройки. Вы можете делать это для вторичного буфера с помощью метода GetEffects. Чтобы лучше слышать изменения при установке, необходимо отключить остальные эффекты, за исключением эха. Добавьте следующую секцию кода сразу после вызова SetEffects:

```
EchoEffect echo = (EchoEffect)sound.GetEffects(0);
EffectsEcho param = echo.AllParameters;
param.Feedback = 1.0f;
param.LeftDelay = 1060.2f;
param.RightDelay = 1595.3f;
param.PanDelay = 1;
echo.AllParameters = param;
```

Здесь мы используем метод GetEffects для создания объекта EchoEffect, определяющего все изменяемые параметры используемого эффекта эха. Теперь проиграв звук с изменениями, выполненными в данном методе, и без них, мы должны сразу почувствовать разницу. Таким образом, мы можем управлять любым загружаемым эффектом.

ИЗМЕНЕНИЕ СВОЙСТВ ЭФФЕКТА

Даже при том, что вызов SetEffects может осуществляться только когда буфер остановлен, вы можете изменять параметры загруженных эффектов в реальном времени, даже при проигрывающем буфере.

Краткие выводы

В этой главе были рассмотрены следующие вопросы.

- Загрузка и проигрывание статических звуков.
  Проигрывание звуков в 3D пространстве.
- Проигрывание звуков с эффектами.

В нашей следующей главе мы ознакомимся с устройствами ввода, рассмотрим возможности управления клавиатурой, мышью и джойстиком.

Глава 15. Управление устройствами ввода

К настоящему моменту мы охватили концепции построения 3D-графики и рассмотрели вопросы использования звука.

Тем не менее, до сих пор не обсуждались темы, связанные с пользовательским интерфейсом. В главе 6, при написании простой игры, управляющей движением автомобиля, мы использовали только клавиатуру, не затрагивая мышь или джойстик. Также мы не рассматривали возможности обратной связи при управлении приложением.

В этой главе мы коснемся интерфейса DirectInput API и узнаем, как с ним работать, чтобы считывать и управлять данными, приходящими с устройств ввода. Разделы этой главы включают.

- Управление с клавиатуры.
- Управление с помощью мыши.
- Управление с помощью джойстика и игровой клавиатуры.
- Работа с обратной связью.

Обнаружение устройств

В первую очередь, чтобы использовать код, который будем обсуждать далее в этой главе, мы должны обеспечить необходимые ссылки на DirectInput. Нам также необходимо добавить ссылку на Microsoft.DirectX.DirectInput и директиву using для этого пространства имен.

Даже если ваш компьютер является автономной системой, в нем имеется по крайней мере два устройства ввода данных: клавиатура и мышь.

Дополняя машину различными USB-устройствами, ставшими достаточно распространенными в настоящее время, мы можем иметь несколько устройств ввода данных, которые мы должны уметь обнаруживать и .распознавать. Если вспомнить, в начале книги мы говорили о классе Manager, который входит в Direct3D. Приложение DirectInput имеет подобный класс, который мы будем использовать для этих задач.

Самая простая вещь, которую мы можем сделать, состоит в том, чтобы обнаружить все имеющиеся в системе устройства. Для этого мы можем использовать свойство devices property в соответствующем классе manager. Вначале необходимо создать и заполнить соответствующим образом разветвленную схему для нашей формы.

Для этой схемы в приложении необходимо добавить некоторые константы ключевых имен:

```
private const string AllItemsNode = "All Items";
private const string KeyboardsNode = "All Keyboard Items";
private const string MiceNode = "All Mice Items";
```

```
private const string GamePadNode = "All Joysticks and Gamepad Items";
private const string FeedbackNode = "All ForceFeedback Items";
```

Опираясь на эти константы, необходимо создать пять различных узлов или папок в дереве разветвленной схемы: одна папка для всех устройств вашей системы, затем по одной секции для мыши, клавиатур, джойстиков и элементов обратной связи. Элементы, находящиеся в папке «All Items», должны быть продублированы в остальных папках.

Вначале необходимо заполнить папку «All Items». Для этого мы создадим функцию LoadDevices, целью которой является заполнение разветвленной схемы элементами, находящимися в системе. Добавьте метод, приведенный в листинге 15.1.

Листинг 15.1. Добавление устройств к разветвленной схеме.

```
public void LoadDevices()
{
   TreeNode allNodes = new TreeNode(AllItemsNode);
   // First get all devices
   foreach(DeviceInstance di in Manager.Devices)
   {
      TreeNode newNode = new TreeNode (string. Format)" (01 - (1} ({2})\
         di.InstanceName,  Manager.GetDeviceAttached(di.InstanceGuid)
         ? "Attached" : "Detached", di.InstanceGuid));
      allNodes.Nodes.Add(newNode);
   }
   treeViewl.Nodes.Add(allNodes);
}
```

Как вы можете видеть, класс DeviceList (производный от класса Devices) возвращает список структур DeviceInstance. Эта структура содержит всю полезную информацию относительно устройств, включая идентификатор GUID (использующийся при создании устройства), название продукта и тип устройства.

Для проверки наличия устройства вы можете использовать и другие методы класса Manager. Вполне возможно иметь в системе и виртуальное устройство «available», которое, в принципе, поддерживается, но на данный момент отсутствует в системе.

И, наконец, мы добавляем каждое найденное в системе устройство в соответствующую папку и добавляем папку в разветвленную схему.

Теперь предположим, что мы хотели найти лишь некоторые типы устройств, например, только клавиатуру? Добавьте код листинга 15.2 в конце метода LoadDevices.

Листинг 15.2. Добавление клавиатуры к разветвленной схеме.

```
// Now get all keyboards
TreeNode kbdNodes = new TreeNode(KeyboardsNode);
foreach(Device!nstance di in Manager.GetDevices(DeviceClass.Keyboard,
   EnumDevicesFlags.AttachedOnly))
{
   TreeNode newNode = new TreeNode(string.Format)"{0} - {1} ({2})",
      di.InstanceName, Manager.GetDeviceAttached(di.InstanceGuid)
      ? "Attached" : "Detached", di.InstanceGuid));
   kbdNodes.Nodes.Add(newNode);
}
treeViewl.Nodes.Add(kbdNodes);
```

Здесь используется новый метод GetDevices класса Manager (вместо знакомого нам Devices), который позволяет определять типы перечисленных устройств. В данном примере мы пытаемся обнаружить объекты keyboards, более того, мы хотим найти только те устройства, которые непосредственно подсоединены к системе. Для определения устройств других типов код остается таким же, только указываются соответствующие значение используемого класса:

```
// Now get all mice
TreeNode miceNodes = new TreeNode(MiceNode);
foreach(DeviceInstance di in Manager.GetDevices(DeviceClass.Pointer,
   EnumDevicesFlags.AttachedOnly))
{
   TreeNode newNode = new TreeNode(string.Format("{0} - {1} ({2})",
      di.InstanceName, Manager.GetDeviceAttached(di.InstanceGuid)
      ? "Attached" : "Detached", di.InstanceGuid)),•
   miceNodes.Nodes.Add(newNode);
}
treeViewl.Nodes.Add(miceNodes);
// Now get all joysticks and gamepads
TreeNode gpdNodes = new TreeNode(GamePadNode);
foreach(DeviceInstance di in Manager.GetDevices(DeviceClass.GameControl,
   EnumDevicesFlags.AUDevices))
{
   TreeNode newNode = new TreeNode(string.Format("(0} - (1} ((21)",
      di.InstanceName, Manager.GetDeviceAttached(di.InstanceGuid)
      ? "Attached" : "Detached", di.InstanceGuid));
   gpdNodes.Nodes.Add(newNode);
}
treeViewl.Nodes.Add(gpdNodes);
```

Обратите внимание, что указатель устройства мыши в классе имеет тип Pointer. Данный класс не подразумевает использование только для устройства мыши, это более общий тип. Экранные указатели Screen pointers, например, относятся к этой же категории.

Теперь мы попробуем определить или найти в системе устройства с обратной связью. Оно не относится к какому-либо определенному типу устройств, это скорее характеристика или особенность, которую мы хотим поддерживать.

Эта проверка также весьма проста:

```
// Now get all Force Feedback items
TreeNode ffNodes = new TreeNode(FeedbackNode);
foreach(DeviceInstance di in Manager.GetDe?ices(DeviceClass.All,
    EnumDevicesFlags.ForceFeeback))
{
    TreeNode newNode = new TreeNode(string.Format("{0} - {1} ({2})",
        di.InstanceName, Manager.GetDeviceAttached(di.InstanceGuid)
        ? "Attached" : "Detached", di.InstanceGuid));
    ffNodes.Nodes.Add(newNode);
}
treeViewl.Nodes.Add(ffNodes);
```

Полная структура кода остается такой же, мы только изменили метод перебора, чтобы перечислить только те устройства, которые поддерживают обратную связь.

ИСПОЛЬЗОВАНИЕ ПАНЕЛИ УПРАВЛЕНИЯ

В классе Manager имеется метод RunControlPanel, который открывает панель управления. Панель управления позволяет установить доступ к настройкам имеющихся в системе устройств.

Использование клавиатуры

Каждое из устройств, использующихся в Directlnput, имеет свой собственный объект Device, подобно тому, как графическая или звуковая карта имеют свои объекты устройств. Количество устройств ввода в Directlnput может быть значительно больше, чем в графических или звуковых картах.

Создание устройства требует наличия идентификатора GUID. Это может быть просто компонент объекта SystemGuid, если мы хотим создать клавиатуру или устройство мыши по умолчанию. Устройства, которые создаются при этом, достаточно универсальны и позволяют извлекать данные из любого устройства, поддерживаемого приложением Directlnput.

Первое устройство, с которого мы начнем, — клавиатура. Во-первых, нам понадобится переменная устройства:

```
private Device device = null;
```

Теперь у нас есть два пути, которые мы можем использовать в Directlnput. Мы можем применить обычный цикл и получить текущее состояние устройства для каждого фрейма, а можем установить Directlnput таким образом, чтобы изменить состояние игры в момент, когда появляется информация об изменении устройства. Приведенный ниже код изначально использует первый механизм. Добавьте следующую функцию инициализации:

```
private bool running = true;
public void Initializelnput()
 {
    // Create our keyboard device
    device = new Device(SystemGuid.Keyboard);
    device.SetCooperativeLevel(this, CooperativeLevelFlags.Background \
       CooperativeLevelFlags.NonExclusive);
    device.Acquire();
    while(running)
     {
       UpdatelnputState();
       Application.DoEvents();
     }
 }
```

Как вы видите, это — полный цикл ввода, вначале которого, используя стандартный модификатор GUID, создается устройство клавиатуры. Можно вспомнить, что в разделе DirectSound мы устанавливали уровень совместного доступа для различных устройств системы. В нашем случае также могут использоваться различные флажки и их сочетания при объявлении уровней совместного доступа, см. таблицу 15.1.

Таблица 15.1. Уровни совместного доступа в **Directlnput**

Флажок	**Описание**
Background	Фоновый доступ. Устройство может использоваться на заднем плане или может быть вызвано в любое время, даже если связанное окно не является активным
Foreground	Активный доступ. Устройство может использоваться только при активном окне, в противном случае, не может быть вызвано

Флажок	Описание
Exclusive	Эксклюзивный режим. Устройство имеет статус монопольного доступа из приложения. Кроме него, никакое другое приложение не может иметь такой же статус доступа, однако неэксклюзивные запросы возможны. Из соображений безопасности на некоторых устройствах исключено совместное использование флажков exclusive и background, например, у клавиатуры и мыши
NonExclusive	Совместный режим. Устройство может быть распределено по многим приложениям и не требует монопольного доступа.
NoWindowsKey	Отключает клавишу «windows key»

Для данного приложения мы можем использовать флажки foreground и non-exclusive. После входа в цикл при выполнении приложения мы переписываем состояние ввода устройства InputState и вызываем обработчик событий DoEvents. Прежде чем записать код метода UpdateInputState, необходимо создать текстовое поле с атрибутами мультистроки и «только для чтения» и установить для него свойство Dock в значении «Fill». Теперь мы можем добавить следующий метод для изменения текстового окна textbox:

```
private void UpdateInputState()
{
   // Check the keys currently pressed first
   string pressedKeys = "Using GetPressedKeys(): \r\n";
   foreach(Key k in device.GetPressedKeysf))
      pressedKeys += k.ToString() + " ";
   textBoxl.Text = pressedKeys;
}
```

Перед запуском этого приложения осталось сделать две вещи. Следует обратить внимание на то, что цикл будет выполняться до тех пор, пока переменная running не получит значение «false», поэтому, если мы хотим, чтобы выполнение приложения закачивалось при закрытии формы, необходимо присвоить переменной running значение «false»:

```
protected override void OnClosed(EventArgs e)
{
   running = false;
}
```

Это может помочь, когда мы вызываем метод инициализации InitializeInput. Изменим основной метод следующим образом:

```
static void Main()
(
   using (Forml frm = new Forml0)
   (
      frm.Show();
      frm.InitializelnputO ;
   I
}
```

Теперь мы можем запустить это приложение. Нажимая и удерживая клавишу в любой момент времени, мы окажемся в текстовом поле. Обратите внимание, что мы уже определили необходимые клавиши, не прибегая к дополнительному коду.

УДЕРЖИВАНИЕ НЕСКОЛЬКИХ КЛАВИШ

Большинство клавиатур могут поддерживать нажатие до пяти клавиш одновременно. Нажатие большего числа клавиш приведет к игнорированию операций.

Помимо описанного метода, использующего игровой цикл, для проверки изменения состояния устройства в Directlnput можно использовать отдельный трэд или поток, работающий по принципу обработчика событий.

Создайте новый метод инициализации ввода, листинг 15.3.

Листинг 15.3. Метод инициализация для Directlnput и Second Thread.

```
private System.Threading.AutoResetEvent deviceUpdated;
private System.Threading.ManualResetEvent appShutdown;
public void InitializelnputWithThreadO
{
   // Create our keyboard device
   device = new Device(SystemGuid.Keyboard);
   device.SetCooperativeLevel(this, CooperativeLevelFlags.Background
      CooperativeLevelFlags.NonExclusive);
   deviceUpdated = new System.Threading.AutoResetEvent(false) ;
   appShutdown = new System.Threading.ManualResetEvent(false);
   device.SetEventNotification(deviceUpdated);
   System.Threading.Thread threadLoop = new System.Threading.Thread)
      new System.Threading.ThreadStart(this.ThreadFunction));
   threadLoop.Start ();
   device.Acquired ;
}
```

Основная предпосылка этого метода такая же, как и в предыдущем, только здесь мы объявили две переменные обработчика событий. Одна из них — AutoResetEvent — для DirectInput, другая — ManualResetEvent, чтобы уведомить трэд, когда приложение закрывается. Нам также понадобится обработчик ThreadFunction, отслеживающий одно из этих событий:

```
private void ThreadFunction()
{
    System.Threading.WaitHandlef]  handles  =
        { deviceUpdated,  appShutdown  };
    // Continue running this thread until the app has closed
    while(true)
    {
        int  index  =  System.Threading.WaitHandle.WaitAny(handles);
        if  (index == 0)
        {
            UpdateInputState();
        }
        else if  (index == 1)
        {
            return;
        }
    }
}
```

Эта функция является паритетной, когда мы имеем дело с многопоточными приложениями. В случае если устройство DirectInput сообщает нам о событии изменения состояния устройства, мы вызываем метод UpdateInputState; если регистрируется другое событие, возвращается функция, которая прерывает поток. Как и в методе с использованием циклов, мы должны сделать еще две вещи, чтобы завершить написание приложения. Необходимо задать обработчик закрытия приложения при регистрации события. Для этого замените перегрузку OnClosed на следующую:

```
protected  override  void  OnClosed(EventArgs  e)
{
    if  (appShutdown  != null)
        appShutdown.Set();
}
```

И мы должны модифицировать нашу основную процедуру, чтобы вызвать новый метод инициализации:

```
static  void  Main()
{
    using  (Forml  fm  =  new  Form())
```

```
        frm.Show() ;
        frm.InitializeInputWithThread();
        Application.Run(frm);
    }
}
```

В большинстве случаев нет необходимости получать список всех нажатых в данный момент клавиш; допустим, нам захотелось обнаружить нажатие определенной клавиши. В этом сценарии мы получаем состояние клавиш и проверяем или сравниваем с тем состоянием, которое ищем. Например, чтобы увидеть, была ли нажата клавиша выхода «ESC», можно сделать следующее:

```
KeyboardState state = device.GetCurrent KeyboardState();
if (state[Key.Escape])
{
    /* Escape was pressed */
}
```

ОТКРЫТИЕ ПАНЕЛИ УПРАВЛЕНИЯ, ЗАВИСЯЩЕЙ ОТ КОНКРЕТНОГО УСТРОЙСТВА

В классе Device имеется метод RunControlPanel, позволяющий открывать панель управления в зависимости от выбранного устройства. Например, при создании клавиатуры этот метод откроет панель со свойствами клавиатуры; для устройства мыши, откроет папку свойств мыши. Возможно внесение изменений в некоторые опции.

Использование устройства мыши

Все устройства DirectInput используют один и тот же класс устройств Devices, так что различия между использованием мыши и клавиатуры весьма незначительные. Необходимо переписать метод создания устройства в коде InitializeInput(), чтобы использовать GUID идентификатор мыши вместо клавиатуры:

```
device = new Device(SystemGuid.Mouse);
```

ЭКСКЛЮЗИВНОЕ ИСПОЛЬЗОВАНИЕ МЫШИ

При работе с мышью в эксклюзивном режиме в некоторых случаях мы можем полностью скрыть курсор мыши. Иногда это востребовано, но как правило, это не желательный эффект. Например, если вы пишете систему меню для вашей игры, которая работает в полно-

экранном режиме, пользователь может использовать это меню только при помощи мыши. Необходимо предусмотреть для этого случая отдельные курсоры.

Для работы приложения нам необходимо изменить часть кода, где мы принимаем данные, поскольку для мыши эта операция теряет смысл. Перепишите метод UpdatelnputState следующим образом:

```
private void UpdatelnputState!)
{
   // Check the mouse state
   MouseState state = device.CurrentMouseState;
   string mouseState = "Using CurrentMouseState: \r\n";
   // Current location of the mouse
   mouseState += string. Format ("{0}x{1}x{2}\r\n", state.X,
      state.Y, state.Z);
   // Buttons
   byte[] buttons = state.GetMouseButtons();
   for(int i = 0; i < buttons.Length; i++)
      mouseState += string.Format("Button {0} (l}\r\n",
         i, buttons[i] != 0 ? "Pressed" : "Not Pressed");
   textBoxl.Text = mouseState;
}
```

Здесь, вместо использования вспомогательной функции для получения текущих данных, мы берем мгновенное состояние (мгновенный отпечаток) мыши. Затем выводится текущая позиция мыши по осям X, Y, и Z (Ось Z обычно относится к колесу или скролу мыши), также как и состояние кнопок.

Запуская приложение теперь, мы обратим внимание, что большую часть времени, «позиция» мыши сообщается как 0x0x0, даже когда мы вращаем мышь вокруг своей оси. Почему же не обновляется пользовательский интерфейс? В действительности, данные мыши сообщаются в относительных единицах оси (относительно последнего фрейма), а не в абсолютных. Если мы хотим сообщить о данных мыши в абсолютных значениях, необходимо сразу после создания устройства изменить строку «свойства» следующим образом:

```
device.Properties.AxisModeAbsolute = true;
```

Теперь при перемещении мыши возвращаемые данные представлены какими-то сумасшедшими цифрами. До того как приложение отследит эти данные, необходимо отследить корректность собственно перемещения мыши в приложении. Большинство приложений использует режим относительных единиц оси, который устанавливается по умолчанию.

Использование игровых клавиатур и джойстиков

В то время как основные принципы использования джойстика, мыши и клавиатуры похожи (они совместно используют тот же самый класс устройств Device), имеются некоторые особенности, касающиеся применения джойстиков, поскольку они в меньшей степени стандартизированы. Мы можем хотя бы в какой-то мере предполагать, что у мыши будут две кнопки, а клавиатура будет иметь по крайней мере 36 клавиш. Формы, размеры и расположение кнопок и осей на джойстиках не предсказуемы. Вы можете иметь один джойстик с двумя осями и двумя кнопками, другой с 3 осями, 10 кнопками, третий подразумевает педаль и т. д.

Не существует также и значения по умолчанию SystemGuid для создания джойстика. Мы должны вначале перечислить джойстики. Перепишите функцию создания устройства, используя листинг 15.4.

Листинг 15.4. Инициализация джойстиков.

```
public bool InitializeInput()
{
   // Create our joystick device
   foreach(DeviceInstance di in Manager.GetDevices(DeviceClass.GameControl,
      EnumDevicesFlags.AttachedOnly))
   {
      // Pick the first attached joystick we see
      device = new Device(di.InstanceGuid);
      break;
   }
   if (device == null) // We couldn't find a joystick
      return false;
   device.SetDataFormat(DeviceDataFormat.Joystick);
   device.SetCooperativeLevel(this, CooperativeLevelFlags.Background |
      CooperativeLevelFlags.NonExclusive);
   device.Properties.AxisModeAbsolute = true;
   device.Acquired();
   while(running)
   {
      UpdateInputState();
      Application.DoEvents ();
   }
   return true;
}
```

Здесь мы используем метод перебора, рассмотренный в начале этой главы, для того, чтобы найти подсоединенный джойстик или игровую

клавиатуру. Поскольку мы не создавали устройство с идентификатором GUID, необходимо сообщить DirectInput о типе устройства, используя метод SetDataFormat. Также необходимо возвратить соответствующее значение в случае, если указанное устройство отсутствует. Перепишем основную функцию, чтобы обработать этот вариант:

```
static void Main()
{
   using (Form1 frm = new Form1())
   {
      frm.Show ();
      if (!frm.InitializeInput())
         MessageBox.Show("Couldn't find a joystick.");
   }
}
```

Современные джойстики обычно имеют несколько осей управления, и диапазон этих осей, как правило, неизвестен. Мы должны обработать все эти оси в диапазоне 10,000 единиц. Вполне возможно, что устройство не поддерживает такое разрешение, но DirectInput будет эмулировать его. Нам необходимо найти оси на джойстике и переопределить диапазон. Добавьте следующий код сразу после создания устройства:

```
// Enumerate any axes
foreach(DeviceObjectInstance doi in device.Objects)
{
   if ((doi.ObjectId & (int)DeviceObjectTypeFlags.Axis) != 0)
   {
      // We found an axis, set the range to a max of 10,000
      device.Properties.SetRange(ParameterHow.ById,
      doi.ObjectId, new InputRange(-5000, 5000));
   }
}
```

Этим способом мы можем перечислить объекты любого устройства. Выполнение данного метода на клавиатуре, например, возвратит объект для каждой клавиши клавиатуры. В нашем случае определяем найденные оси, затем изменяем свойства данного объекта (с учетом номера ID) таким образом, чтобы гарантировать соответствие указанным диапазонам. И последнее, что мы должны сделать, — переписать пользовательский интерфейс:

```
private void UpdateInputState()
{
   // Check the joystick state
```

```
    JoystickState state = device.CurrentJoystickState;
    string joyState = "Using JoystickState: \r\n";
    joyState += string.Format("{0}x{l}",
       state.X, state.Y);
    textBoxl.Text = joyState;
}
```

Мы могли бы, используя предшествующее перечисление, обнаружить все элементы джойстика и соответственно изменить пользовательский интерфейс, но это не даст нам ничего нового.

В данном примере, использующем метод UpdatelnputState, мы модифицируем только X и Y оси, которые присутствуют в большинстве джойстиков.

УПРАВЛЕНИЕ ДИАПАЗОНОМ

Игровые клавиатуры (и цифровые джойстики) обычно перескакивают от одного экстремального значения диапазона к другому. В примере, рассмотренном ранее, когда игровая клавиатура находилась «в покое», ее значение было бы равно «0». Если бы вы нажали левую кнопку, ось X переместилась бы мгновенно в минимальное значение диапазона, то же самое и с правой кнопкой — резкое перемещение в максимальное значение без каких-либо промежуточных значений.

Существует два типа джойстиков, цифровые и аналоговые. Цифровой джойстик мы только что описали. В аналоговых джойстиках диапазоны, как правило, встроены. Хороший пример этого — штурвалы управления полетом. В этих джойстиках имеется широкий диапазон перемещений по осям, что позволяет достаточно четко отслеживать эти перемещения.

Устройства обратной связи

Использование обратной связи в вашем приложении позволяет ощутить наибольшее погружение в игровой процесс. Например, врезаясь в дерево на скорости 100 миль в час, при наличии джойстика с такой опцией вы можете реально ощутить удар, передающийся в руку.

Первое, что хотелось бы сразу отметить при упоминании эффектов обратной связи Force Feedback (дословно обратная связь по усилию), это то, что мы должны иметь эксклюзивный доступ к устройству. К счастью, для таких устройств используется установка по умолчанию режима эксклюзивного доступа и фонового режима.

Итак, создадим устройство для этого примера (аналогично процедуре для джойстика). Перепишите метод Initializelnput, как в листинге 15.5.

Листинг 15.5. Инициализация устройства с обратной связью.

```
private ArrayList effectList = new ArrayList();
public bool InitializeInput()
{
   // Create our joystick device
   foreach(DeviceInstance di in Manager.GetDevices(DeviceClass.GameControl,
      EnumDevicesFlags.AttachedOnly | EnumDevicesFlags.ForceFeeback))
   {
      // Pick the first attached joystick we see
      device = new Device(di.InstanceGuid);
      break;
   }
   if (device == null) // We couldn't find a joystick
      return false;
   device.SetDataForaat(DeviceDataFormat.Joystick);
   device.SetCooperativeLevel(this, CooperativeLevelFlags.Background |
      CooperativeLevelFlags.Exclusive);
   device.Properties.AxisModeAbsolute = true;
   device.Acquire ();
   // Enumerate any axes
   foreach(DeviceObjectInstance doi in device.Objects)
   {
      if ((doi.ObjectId & (int)DeviceObjectTypeFlags.Axis) != 0)
      {
         // We found an axis, set the range to a max of 10,000
         device.Properties.SetRange(ParameterHow.ById,
         doi.Objectld, new InputRange(-5000, 5000));
      }
   }
   II Load our feedback file
   EffectList effects = null;
   effects = device.GetEffects(@"..\..\idling.ffe",
      FileEffectsFlags.ModifylfNeeded);
   foreach(FileEffect fe in effects)
   {
      EffectObject myEffect = new EffectObject(fe.EffectGuid, fe.EffectStruct,
         device);
      myEffeet.Download();
      effectList.Add(myEffect);
   }
   while(running)
   {
      UpdateInputState();
      Application.DoEvents ();
   }
   return true;
}
```

Как вы видите, здесь имеется несколько изменений. Мы установили эксклюзивный доступ и переопределили метод перебора returnforce для устройства обратной связи. Кроме того, мы установили еще раз диапазоны и определили эффекты обратной связи.

Следует отметить, что мы можем создавать эффект обратной связи из файла. В этом случае каждый эффект может быть представлен в виде массива отдельных эффектов обратной связи. Код, включенный в CD диск (и в нашем тексте) будет использовать файл idling.ffe, поставляемый с DirectX SDK. Также на CD диске вы можете найти копию исходника.

После того как мы загрузили список эффектов из файла, мы просматриваем каждый отдельный эффект обратной связи и создаем объект эффекта. Далее мы загружаем этот эффект в фактическое устройство таким образом, чтобы устройство могло использовать его и проверять его состояние, и наконец добавляем этот эффект к локальному списку массива.

Теперь, когда мы имеем набор загруженных эффектов, мы можем использовать их в нашем устройстве. Для этого необходимо добавить метод, который определяет, проигрывается ли данный эффект в настоящее время или нет, и если нет — запустить его. Это будет гарантировать проигрывание эффекта при запущенном приложении.

```
private void PlayEffects ()
 {
    // See if our effects are playing.
    foreach(EffectObject myEffect in effectList)
    {
       if (ImyEffect.EffectStatus.Playing)
       {
          // If not, play them
          myEffect.Start (1, EffectStartFlags.NoDownload);
       }
    }
 }
```

Целесообразно также добавить вызов PlayEffects в наш метод UpdateInputState.

ИСПОЛЬЗОВАНИЕ РЕДАКТОРА УСИЛИЯ Force Editor

Утилита Force Editor, которая поставляется с DirectX SDK, может использоваться для создания любого эффекта обратной связи по желанию. Помимо этого возможно создавать и редактировать эти эффекты вручную в нашем коде, затем сохранять их в файл и использовать в дальнейшем.

Краткие выводы

В этой главе мы рассмотрели.

Управление с клавиатуры.

Управление с помощью мыши.

Управление с помощью джойстика и игровой клавиатуры.

Устройства обратной связи Force Feedback.

В последующих главах мы рассмотрим возможности Direct3D для двухмерной графики, а также добавление сетевых возможностей.

ЧАСТЬ V

2D ГРАФИКА

Глава 16. Приложение Direct3D для 2D-графики

Глава 17. Использование DirectDraw для рендеринга 2D-графики

Глава 16. Приложение Direct3D для 20-графики

В этой главе мы рассмотрим вопросы программирования и рендеринга 20-графики с помощью Direct3D. Мы коснемся следующих тем.

- Создание полноэкранного устройства.
- Использование класса Sprite.
- Рендеринг спрайтов.

Создание полноэкранного устройства отображения

Зачастую рендеринг громоздких трехмерных сцен «сложен» по опре-" делению. Несмотря на то, что большинство современных компьютерных игр используют богатую трехмерную графику, существует множество игровых сюжетов, где нет необходимости использовать «причуды» 3D-графики. Поэтому имеет смысл создание таких игр в 2D-исполнении.

Учитывая то, что Direct3D может отображать достаточно сложные трехмерные сцены, надо полагать, что рендеринг 2D-объектов не будет для нас чем-то чрезмерно громоздким.

Создание полноэкранного устройства не слишком отличается от создания устройств, работающих в оконном режиме, за исключением некоторых особенностей.

Создайте форму, которая будет использоваться нами в дальнейшем при визуализации 20-графики. Как обычно, перед созданием устройства необходимо добавить директиву using, объявить переменную устройства и установить стиль окон.

Как только проект создан, необходимо объявить две константы, определяющие полноэкранный размер (в нашем случае 800x600):

```
public const int ScreenWidth = 800;
public const int ScreenHeight = 600;
```

Эти константы определяют ширину и высоту полноэкранного окна, которое мы будем использовать для отображения. Теперь мы напишем метод инициализации полноэкранного устройства, приведенный в листинге 16.1.

Листинг 16.1. Инициализация полноэкранного устройства

```
public void InitializeGraphics ()
 {
```

```
    // Set our presentation parameters
    PresentParameters presentParams = new PresentParameters() ;
    presentParams.SwapEffeet = SwapEffeet.Discard;
    ' // Start up full screen
    Format current = Manager.Adapters[0].CurrentDisplayMode.Format;
    if (Manager.CheckDeviceType(0, DeviceType.Hardware, current, current,
    false))
    {
       // Perfect, this is valid
       presentParams.Windowed = false;
       presentParams.BackBufferFormat = current;
       presentParams.BackBufferCount = 1;
       presentParams.BackBufferWidth = ScreenWidth;
       presentParams.BackBufferHeight = ScreenHeight;
    }
    else
    {
       presentParams.Windowed = true;
    }
    // Create our device
    device = new Device(0, DeviceType.Hardware, this,
       CreateFlags.SoftwareVertexProcessing, presentParams);
}
```

Начало этого метода должно быть нам знакомо. Поскольку в данном разделе мы рассматриваем 2D-объекты, нет никаких причин объявлять буфер глубины в приведенном методе инициализации.

Можно предположить, что текущий формат режима дисплея для заданного по умолчанию адаптера является подходящим для создаваемого полноэкранного устройства. Для того чтобы принять это предположение, целесообразно вызвать метод проверки CheckDeviceType. Если этот тип устройства поддерживается, измените параметры описания, чтобы определить полноэкранное устройство.

Параметр Windowed устанавливается в значение «false», так как устройство будет работать в полноэкранном режиме. Для отображения в полноэкранном режиме мы будем применять вторичный буфер, поэтому ширина и высота вторичного буфера должны быть установлены в соответствии с заявленными константами размера экрана.

ОБРАБОТКА СОБЫТИЙ ОТКАЗОВ ПРИ СОЗДАНИИ УСТРОЙСТВА

Предположим (хотя это маловероятно), что при создании устройства произойдет отказ и, несмотря на то, что формат устройства перед созданием был проверен, вторичный буфер не будет создан.

Формат 800x600, выбранный для нашего примера, является достаточно распространенным, и есть все основания полагать, что данный формат будет поддерживаться. Вопросы определения подходящего режима дисплея рассматривались в главе 2.

ИСПОЛЬЗОВАНИЕ ПОЛНОЭКРАННОГО УСТРОЙСТВА

Полноэкранный режим не является особенностью отображения 2D-графики. Используя подход, рассмотренный в листинге 16.1, можно создать практически любое, в том числе и трехмерное, полноэкранное устройство. В предыдущих главах, рассматривая объекты 3D-графики, мы опирались в основном на оконный режим по причине большой трудоемкости в отладке кода для полноэкранного режима.

В случае отказа при создании устройства можно всегда вернуться к оконному режиму, управляя соответствующим значением в структуре параметров представления. После заполнения данной структуры мы можем приступить к созданию устройства.

Итак, мы создаем полноэкранное устройство. Вполне естественно, что у нас возникает желание предусмотреть простую процедуру завершения работы приложения. Для этого запишем известный нам по прошлым главам обработчик события нажатия клавиши, добавив его к нашему коду:

```
protected override void OnKeyUp(KeyEventArgs e)
 {
    if (e.KeyCode == Keys.Escape) // Quit
       this.Closed;
    base.OnKeyUp (e);
 }
```

Рендеринг спрайтов

Приложение Direct3D уже содержит объекты текстурирования или текстуры, которые можно уподобить спрайтам. Однако, визуализация текстур подразумевает некий примитив или поверхность, на которую данная текстура должна отображаться. Класс Sprite позволяет выполнять эту процедуру автономно и независимо от использования примитивов. Добавьте следующие переменные для нашего приложения:

```
private Texture spriteTexture;
private Sprite sprite;
private Rectangle textureSize;
```

Эти переменные позволяют сохранить спрайт и текстуру, которая будет использоваться для рендеринга спрайта. Для правильного выполнения данной процедуры необходимо знать размер сохраняемой текстуры (переменная textureSize).

Таким образом, необходимо инициализировать эти переменные, добавив следующий код в конце метода InitializeGraphics, сразу после создания устройства:

```
// Create our texture
spriteTexture = TextureLoader.FromFile(device, @"..\..\logo.tga") ;
using (Surface s = spriteTexture.GetSurfaceLevel(0))
{
   SurfaceDescription desc = s.Description;
   textureSize = new Rectangle(0, 0, desc.Width, desc.Height);
}
sprite = new Sprite(device);
```

Файл logo.tga находится на CD диске. После создания текстуры необходимо определить ее размер.

Мы находим поверхность, на которую будет отображаться текстура, и, используя ее описание, устанавливаем размер текстуры. Затем мы создаем пример класса Sprite.

Класс Sprite позволяет, используя различные текстуры, рисовать любое количество спрайтов. Для этого, вместо того чтобы копировать наш класс Sprite, мы создадим новый класс, позволяющий обработать множественные спрайты. Добавьте класс, приведенный в листинге 16.2.

Листинг 16.2. Добавление класса Sprite

```
public class GraphicsSprite
{
    // static data for our sprites
    private static readonly Vector3 Center = new Vector3(0, 0, 0);
    private static readonly Random rnd = new Random();
    // Instance data for our sprites
    private Vector3 position;
    private float xUpdate = 1.4f;
    private float yUpdate = 1.4f;
    public GraphicsSprite(int posx, int posy)
    {
       position = new Vector3(posx,posy,1);
       xUpdate += (float)rnd.NextDoublef);
       yUpdate += (float)rnd.NextDoublef);
    }
```

```
    public void Draw(Sprite sprite, Texture t, Rectangle r)
    {
        sprite.Draw(t, r, Center, position, Color.White);
    }
}
```

При желании мы можем вращать спрайты (даже если приложение не предусматривает эту операцию).

Когда мы рисуем спрайт (sprite.Draw), одним из параметров этой функции, используемым для задания вращения, является центр спрайта. Проще сохранить это значение здесь, вместо того чтобы каждый раз при вызове функции создавать новый вектор.

В данной процедуре мы также добавляем движению спрайтов элемент случайности (параметр Random).

Для каждого спрайта устанавливаются параметры, отвечающие за текущую позицию спрайта и его скорость перемещения.

Различные значения скорости по координатам X и Y позволяют спрайту перемещаться не только под углом в 45°. Объект имеет заданные начальные значения скорости по осям X и Y, которые затем изменяются случайным образом.

Параметр цвета в данном методе используется для придания оттенка отображаемой текстуре.

Белый цвет (как в данном примере) позволит отобразить текстуру с естественным оттенком, а использование красного добавит текстуре красный оттенок.

Используя общий класс Sprite, можно переписать код и включить прорисовку спрайтов.

Перед этим необходимо добавить новую переменную для записи отображаемых спрайтов:

```
System.Collections.ArrayList ar = new System.Collections.ArrayList();
```

Здесь мы задаем пустой массив класса ArrayList, чтобы затем добавить к нему несколько спрайтов.

Для этого запишем следующий код в метод инициализации сразу после создания спрайта:

```
// Add a few sprites
ar.Add(new GraphicsSprite(0, 0));
ar.Add(new GraphicsSprite(64,128));
ar.Addfnew GraphicsSprite(128,64));
ar.Addfnew GraphicsSprite(128,128));
ar.Addfnew GraphicsSprite(192,128));
ar.Addfnew GraphicsSprite(128,192));
ar.Addfnew GraphicsSprite(256,256));
```

По желанию можно изменять количество и местоположение добавляемых спрайтов.

Имея коллекцию спрайтов, мы можем попробовать отобразить их на экране.

Для этого в процедуре OnPaint добавьте после вызова BeginScene следующий код рендеринга спрайтов:

```
// Begin drawing our sprites with alpha blend
sprite.Begin(SpriteFlags.None);
// Draw each sprite in our list.
foreach(GraphicsSprite gs in ar)
   gs.Draw(sprite, spriteTexture, textureSize);
// Notify Direct3D we are done drawing sprites
sprite.End();
```

Функция Begin начинает процедуру рендеринга, затем для каждого спрайта вызывается метод Draw. Метод заканчивается обязательным оператором End.

Теперь мы можем запустить приложение.

Увы, мы видим, что наши спрайты не перемещаются, несмотря на то, что мы задали скорость перемещения.

Это очевидно, поскольку в нашей программе не задан метод обновления спрайтов.

Нечто подобное мы уже проходили в первых главах, когда пытались задать вращение элементарному треугольнику.

Добавьте в класс GraphicsSprite метод Update, приведенный в листинге 16.3.

Листинг 16.3. Обновление спрайтов.

```
public void Update(Rectangle textureSize)
{
   // Update the current position
   position.X += xUpdate;
   position.Y += yUpdate;
   // See if we've gone beyond the screen
   if (position.X > (Forml.ScreenWidth - textureSize.Width))
   {
      xUpdate *= -1;
   }
   if (position.Y > (Forml.ScreenHeight - textureSize.Height))
   {
      yUpdate *= -1;
   }
```

```
// See if we're too high or too the left
if (position.X < 0)
{
   xUpdate *= -1;
)
if (position.Y < 0)
{
   yUpdate *= -1;
 }
```

В рассмотренном методе изменение положения спрайта определяется двумя отдельными переменными скорости.

Затем проверяется местоположение спрайта относительно границ экрана, и, если спрайт коснулся границы экрана, направление перемещения изменяется на противоположное, и объект как бы отражается от края экрана.

Для вызова этого метода добавьте следующий код в начале процедуры OnPaint:

```
// Before we render each sprite, make
// sure we update them
foreach(GraphicsSprite gs in ar)
   gs.Update(textureSize);
```

Функция Update обращается к каждому из имеющихся спрайтов. Теперь, запустив приложение, мы увидим объекты, перемещающиеся на экране в различных направлениях.

ИСПОЛНЕНИЕ СПРАЙТОВ С ПАРАМЕТРОМ ПРОЗРАЧНОСТИ ЦВЕТА «АЛЬФА»

При использовании текстуры, хранящейся на CD диске, можно заметить белую прозрачную окантовку текстуры. Эта текстура содержит параметр «альфа» — прозрачность объекта. Использование данного параметра несложно: вместо

```
sprite.Begin(SpriteFlags.None);
```

напишите:

```
sprite.Begin(SpriteFlags.AlphaBlend);
```

На рис. 16.1 показан результат визуализации текстуры, использующей параметр прозрачности «альфа».

Рис. 16.1. Перемещающиеся спрайты

Анимация спрайтов

В основном, при анимации 2D-спрайтов отображаются не просто статические картинки, а скорее, последовательности изображений. В принципе, основной механизм анимации спрайтов такой же, как и для статических изображений. Потгому в качестве отправной точки мы возьмем копию программы, написанной для перемещения статических объектов.

Главные различия между данным приложением и предыдущим заключаются во вспомогательном классе, используемом для рендеринга спрайтов. Процедура инициализации осталась практически той же самой, только в исходнике кода поменялось имя файла «sprites.tga», чтобы подчеркнуть то, что он включает в себя множественные спрайты. Так как вместо одного статичного мы будем отображать на экране большое количество небольших анимированных объектов, необходимо случайным образом задать им первоначальное положение. Таким образом, перепишем конструктор для класса Sprite следующим образом:

```
public  GraphicsSprite()
(
    position = new Vector3(rnd.Hext(Forml.ScreenWidth-SpriteSize),
        rnd.Hext(Forml.ScreenHeight-SpriteSize), 1);
    xUpdate += (float) rnd.UextDouble();
    yUpdate += (float)rnd.UextDouble(),
    column = rnd.Next(NumberSpritesCol);
```

```
        row = rnd.Next(NumberSpritesRow);
        if ((column % 3) == 0)
            xUpdate *=.-1;
        if ((row % 2) == 0)
            yUpdate *= -1;
    }
```

Позиция спрайта на экране задается следующим образом. Определяется разность между шириной и высотой экрана (ScreenWidth и ScreenHeight) и размером спрайта по ширине и высоте соответственно, В результате функции случайных чисел возвращаются целые числа, лежащие в пределах от 0 до этих значений. Однако, мы до сих пор не задали размер спрайта. Размер рамки используемого в нашем примере спрайта составляет 50x45. Теперь мы можем установить эти константы:

```
private const int SpriteSizeWidth = 50;
private const int SpriteSizeHeight = 45;
```

Определив первоначальную позицию, можно с помощью функции случайных чисел задать и скорость, по желанию умножив значение скорости на некий коэффициент для более быстрого перемещения спрайтов. Спрайты в этом примере сформированы серией образцов размера 50x45, последовательно расположенных в файле, состоящем из пяти спрайтов в каждой из шести строк. Сохраните эту информацию, используя константы:

```
private const int NumberSpritesRow = 6;
private const int NumberSpritesCol = 5;
```

Также необходимо объявить переменные, которые будут использоваться для определения текущего фрейма:

```
private int column = 0;
private int row = 0;
```

Теперь, когда мы изменили конструктор класса Sprite, необходимо переписать последние строки в методе инициализации, чтобы добавить наши спрайты:

```
// Add a few sprites
for (int i = 0; i < 100; i++)
    ar.Add (new GraphicsSprite() ) ;
```

Число спрайтов можно при желании изменять.

Теперь, зная размеры спрайта, мы должны изменить метод обновления Update, см. листинг 16.4.

Листинг 16.4. Обновление анимированных спрайтов.

```
public void Updated()
 {
    // Update the current position
    position.X += xUpdate;
    position.Y += yUpdate;
    // See if we've gone beyond the screen
    if (position.X > (Forml.ScreenWidth - SpriteSizeWidth))
    {
       xUpdate *= -1;
    }
    if (position.Y > (Forml.ScreenHeight - SpriteSizeHeight)
    {
       yUpdate *= -1 :

    // See if we're too high or too the left
    if (position.X < 0)
    {
       xUpdate *= -1;
    }
    if (position.Y < 0)
    {
       yUpdate *= -1;
    )
    II Now update the column
    column++;
    if (column >= NumberSpritesCol)
    {
       row++;
       column = 0;
    }
    if (row >= NumberSpritesRow)
    {
       row = 0;
    }
 }
```

Начало этого метода такое же, как и в предыдущем варианте, только теперь мы используем размер спрайта вместо входящего параметра размера текстуры.

Далее выполняется последовательный перебор спрайтов по столбцам и строкам и возврат в начало после прохождения всего набора спрайтов.

Теперь, когда мы изменили метод обновления, можно убрать параметр из обращения к методу Update:

```
foreach(GraphicsSprite gs in ar)
   gs. Updated;
```

Последнее, что мы должны сделать, изменить метод рисования Draw и, соответственно, оформить вызов этой процедуры. Перепишите метод Draw следующим образом:

```
public void Draw(Sprite sprite, Texture t)
 {
    sprite.Draw(t, new Rectangle(column * SpriteSizeWidth, row *
       SpriteSizeHeight,
       SpriteSizeWidth, SpriteSizeHeight), Center, position, Color.White);
 }
```

Легко заметить, что в списке параметров вместо размера появился полигон спрайта, рассчитанный с помощью сохраненных данных объекта анимации. Теперь мы можем переписать процедуру OnPaint для вызова метода Draw:

```
foreach(3raphicsSprite gs in ar)
   gs.Draw(sprite, spriteTexture);
```

Запустив приложение, мы увидим на экране движущиеся анимированные спрайты, как на **рис.** 16.2.

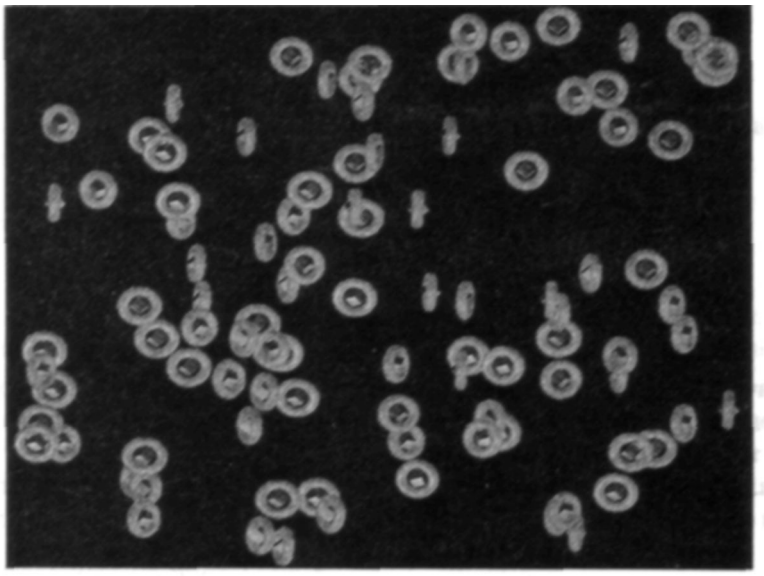


Рис. 16.2. Анимированные спрайты

Краткие выводы

В данной главе мы рассмотрели.

- Использование полноэкранного режима.
- Рендеринг спрайтов.
- Рендеринг анимированных спрайтов.

В следующей главе мы рассмотрим использование в этих же целях DirectDraw вместо Direct3D.

Глава 17. Использование DirectDraw для рендеринга 2D-графики

Одним из недостатков использования Direct3D для 2D-приложений являются аппаратные ограничения. Direct3D более сложен, чем DirectDraw, и требует большего количества ресурсов. В связи с этим в некоторых системах целесообразно использовать устройство DirectDraw. Более того, это может касаться и современных графических плат. При имеющихся аппаратных ограничениях DirectDraw может решить проблему, связанную с ресурсами системы.

В этой главе мы рассмотрим использование DirectDraw для 2D-графики, включая.

Использование полноэкранного режима.

- Рендеринг спрайтов.
- Рендеринг анимированных спрайтов.

Создание полноэкранного устройства DirectDraw

Прежде, чем создать устройство DirectDraw, необходимо выполнить следующие действия:

1. Создать новый проект.
2. Добавить ссылки на Microsoft.DirectX.dll и Microsoft.DirectDraw.dll.
3. Включить новую директиву using для Microsoft.DirectX.DirectDraw в главном файле кода.
4. Установить стиль окна как Opaque (непрозрачный) и AllPaintingln-WmPaint, подобно тому, как это делалось для 3D-приложений.
5. Добавить индивидуальную переменную для DirectDraw устройства.

Практически все эти пункты мы выполняли в наших 3D-приложениях, различия могут быть только в добавляемых ссылках.

Следует отметить, что в DirectDraw для отображения 2D-объектов вместо текстур используются поверхности (surfaces). По сути, поверхность — это сохраненные данные полигонального изображения. При отображении отдельного спрайта в полноэкранном устройстве нам понадобятся три поверхности, которые мы добавляем к приложению:

```
private Surface primary = null;
private Surface backBuffer = null;
private Surface sprite = null;
```

Очевидно, что поверхность sprite является объектом рендеринга сохраненного изображения. Мы могли бы отображать нашу поверхность,

привязываясь непосредственно к экрану, но это может привести к разрыву изображений при отображении нескольких сцен. По этой причине в DirectDraw необходимо выполнять всю процедуру рендеринга во вторичном буфере с последующим копированием в первичный (если вспомнить, в Direct3D данные действия выполнялись бы автоматически).

Прежде чем начать, требуется написать процедуру обработки нажатия клавиши выхода «Escape». Добавим обработчик этого события в наше полноэкранное приложение:

```
protected override void OnKeyUp(KeyEventArgs e)
{
   if (e.KeyCode == Keys.Escape) // Quit
      this.Closed ;
   base.OnKeyUp (e);
}
```

Теперь, как обычно записываем метод инициализации, см. листинг 17.1.

Листинг 17.1. Инициализация DirectDraw устройства.

```
public void InitializeGraphics()
{
   SurfaceDescription description = new SurfaceDescription();
   device = new Device() ;
   // Set the cooperative level.
   device.SetCooperativeLevel(this,
   CooperativeLevelFlags.FullscreenExclusive);
   // Set the display mode width and height, and 16 bit color depth.
   device.SetDisplayMode(ScreenWidth, ScreenHeight, 16, 0, false);
   // Make this a complex flippable primary surface with one backbuffer
   description.SurfaceCaps.PrimarySurface =
      description.SurfaceCaps.Flip =
      description.SurfaceCaps.Complex = true;
   description.BackBufferCount = 1;
   // Create the primary surface
   primary = new Surface(description, device);
   SurfaceCaps caps = new SurfaceCaps!);
   caps.BackBuffer = true;
   // Get the backbuffer from the primary surface
   backBuffer = primary.GetAttachedSurface(caps) ;
   // Create the sprite bitmap surface,
   sprite = new Surfaced". .\. .\logo.bmp",
      new SurfaceDescriptionO, device);
   // Set the colorkey to the bitmap surface.
```

```
        // which is what the CoiorKey struct is initialized to.
        ColorKey ck = new CoiorKey();
        sprite. SetColorKey (ColorKeyFlags.SourceDraw, ck);
    }
```

Как вы можете видеть, этот метод несколько более сложен, чем вариант для Direct3D. После создания устройства DirectDraw мы вызываем процедуру установки уровней доступа к различным ресурсам платы SetCooperativeLevel. Поскольку для DirectDraw мы используем полноэкранное приложение, нет никакой необходимости в совместном использовании ресурсов (устанавливаем эксклюзивный доступ FullScreenExclusive).

Далее устанавливается режим отображения, первые два параметра — ширина и высота экранного окна. Третий параметр определяет насыщенность цвета, в нашем случае используется 16-разрядный формат. Использование 32-разрядного параметра цвета позволяет формировать более богатые цвета, но и требует при этом большего количества ресурсов. Четвертый параметр — частота обновления монитора, для установки по умолчанию используется значение «О». Последний параметр применяется редко, он определяет использование стандартного режима VGA.

После создания устройства и определения режимов отображения формируются поверхности. Вначале создается первичная поверхность, точнее, ее описание. В нашем случае описание первичной поверхности включает в себя опции PrimarySurface, Flip и Complex. Поверхность со свойством Complex может иметь другие присоединенные поверхности, и в нашем случае прикрепленная поверхность будет являться содержимым вторичного буфера.

Теперь, имея соответствующее описание, создается первичная поверхность. Вызывая метод GetAttachedSurface, мы можем определить или задать поверхность вторичного буфера на базе первичной поверхности.

И последним действием в этой процедуре является создание поверхности спрайта. Код, записанный на CD диске, использует для этой цели файл dx5 logo.bmp. Поскольку поверхность создается из файла, нам не понадобится дополнительная информация об этой поверхности (поле описания поверхности останется пустым).

ПАРАМЕТР ПРОЗРАЧНОСТИ В DIRECTDRAW

DirectDraw не поддерживает параметр прозрачности цвета «альфа». В варианте нашего приложения для Direct3D мы просто использовали соответствующую технику для придания фону нашего спрайта свойства прозрачности. В DirectDraw для этого используется так называемый ключевой цвет (color key), который делает один из цветов прозрачным. Правда такая схема имеет и недостатки, одним из

которых является невозможность использования этого цвета в изображении спрайта.

В нашем случае изображение имеет черный цвет фона, который и будет использоваться в качестве ключевого (для этого значение ключевого цвета устанавливается нулевым).

Необходимо отметить, что в процедуре инициализации имеется еще несколько констант, которые мы не объявили:

```
public const int ScreenWidth = 800;
public const int ScreenHeight = 600;
private static readonly Rectangle SpriteSize =
   new Rectangle(0,0,256,256);
```

Ширина и высота экрана встречались и раньше. Поскольку размер изображения, загружаемого из файла и используемого в качестве спрайта, известен, проще его сохранить сейчас, чем определять в дальнейшем. Размер изображения можно также определить, используя описание поверхности (параметры Width и Height).

Теперь мы должны добавить вызов процедуры инициализации из нашего основного метода:

```
static void Main()
{
   using (Forml frm = new Forml())
   (
      // Show our form and initialize our graphics engine
      frm.Show();
      frm.InitiaiizeGraphics();
      Application.Run(frm);
```

Для обработки данных спрайта в нашем приложении нам понадобится общий класс:

```
public class GraphicsSprite
(
   // static data for our sprites
   private static readonly Random rnd = new Random!);
   // Instance data for our sprites
   private int xPosition = 0;
   private int yPosition = 0;
   private float xUpdate = 1.4f;
```

```
    private float yUpdate = 1.4 f;
    /// <summary>
    /// Constructor for our sprite
    /// </summary>
    /// <param name="posx">Initial x position</param>
    /// <param name="posy">Initial y position</param>
    public GraphicsSprite(int posx, int posy)
    {
        xPosition = posx;
        yPosition = posy;
        xUpdate += (float)rnd.NextDouble();
        yUpdate += (float)rnd.NextDouble();
    }
}
```

В этом случае класс сохраняет текущую позицию левого верхнего угла спрайта. Обратите внимание на то, что вместо используемого в Direct3D параметра Vector3 здесь сохраняются два отдельных целочисленных значения. Целочисленные значения здесь вполне корректны и предпочтительны, поскольку DirectDraw использует экранные координаты устройства.

Третий параметр (параметр глубины) опускается, поскольку в приложении DirectDraw он не используется.

Указанный класс будет сохранять скорость для каждого направления перемещаемых по экрану спрайтов, используя начальное значение и случайное приращение скорости спрайта.

Теперь мы должны переписать наш класс Sprite, добавив процедуры обновления и рисования, см. листинг 17.2.

Листинг 17.2. Обновление и рисование спрайтов.

```
public void Draw(Surface backBuffer, Surface spriteSurface,
    Rectangle spriteSize)
{
    backBuffer.DrawFast(xPosition, yPosition, spriteSurface, spriteSize,
        DrawFastFlags.DoNotWait | DrawFastFlags.SourceColorKey);
}
public void Update(Rectangle spriteSize)
{
    // Update the current position
    xPosition += (int)xUpdate;
    yPosition += (int)yUpdate;
    // See if we've gone beyond the screen
    if (xPosition > (Forml.ScreenWidth - spriteSize.Width))
    {
        xUpdate *= -1;
```

```
if (yPosition > (Forml.ScreenHeight - spriteSize.Height))
{
    yUpdate *= -1;
    }
// See if we're too high or too the left
if (xPosition < 0) ^
{
    xUpdate *= -1;
    }
if (yPosition < 0)
{
    yUpdate *= -1;
    }
}
```

РАЗЛИЧИЯ МЕЖДУ ВЫЗОВАМИ DRAW И DRAWFAST

При использовании аппаратной акселерации (что имеет место в большинстве современных графических плат) между методами Draw и DrawFast нет никакого различия. Однако, в случае программной реализации метод DrawFast приблизительно на 10 % быстрее, чем метод Draw, правда, ценой надежности и устойчивости. Метод Draw является более гибким и позволяет реализовать различные операции по отношению к отображаемым спрайтам. Для простых же операций рисования целесообразно использовать метод DrawFast.

Вызов процедуры рисования вызывает поверхность из вторичного буфера, куда будет отображаться спрайт, а также поверхность спрайта и ее размер. Процедура DrawFast просто рисует спрайт в указанном положении, используя весь спрайт в качестве исходного материала. Используемые флажки сообщают DirectDraw, что он «не должен ожидать» окончания процедуры рисования, и что для включения параметра прозрачности необходимо использовать цветовой ключ (который был установлен при создании спрайта).

Приведенный в листинге 17.2 метод Update для DirectDraw идентичен аналогичному методу для Direct3D. Позиция и скорость перемещения спрайта связаны между собой, а направление перемещения изменяется на обратное в случае приближения спрайта к краю экрана. Теперь, полностью описав класс Sprite, мы должны обработать все имеющиеся в нашем приложении спрайты. Будем использовать тот же самый метод, который использовался для Direct3D. Добавьте следующую переменную:

```
System.Collections.ArrayList ar = new System.Collections.ArrayList();
```

Теперь мы можем добавить несколько спрайтов к нашей коллекции, используя тот же код, что и раньше. Для этого добавьте следующие строки в конце метода инициализации:

```
// Add a few sprites
ar.Add(new GraphicsSprite(0,0) );
ar.Add(new GraphicsSprite (64,128));
ar.Addfnew GraphicsSprite(128, 64));
ar.Add(new GraphicsSprite(128,128)) ;
ar.Addfnew GraphicsSprite(192,128)) ;
ar.Add(new GraphicsSprite(128,192));
ar.Addfnew GraphicsSprite(256,256)) ;
```

Осталось лишь добавить процедуру визуализации. Механизм рендеринга остался прежним, поэтому мы можем добавить следующую перегрузку в класс windows form:

```
protected override void OnPaint(System.Windows.Forms.PaintEventArgs e)
{
    Microsoft.DirectX.DirectXException.IgnoreExceptions();
    foreach(GraphicsSprite gs in ar)
        gs.Update(SpriteSize);
    backBuffer.ColorFill(O);
    foreach(GraphicsSprite gs in ar)
        gs.Draw(backBuffer, sprite, SpriteSize);
    primary.Flip(backBuffer, FlipFlags.DoNotWait);
    this.Invalidated ;
}
```

Первое, что бросается в глаза в этом методе, — выключение параметра обработки исключительных ситуаций в Управляемом DirectX. Этот момент мы обсудим позже, сейчас же следует отметить, что необходимо пользоваться этим с достаточной осторожностью.

Теперь, когда мы, используя сохраненный размер, изменили каждый спрайт, очищаем вторичный буфер (в нашем случае для этого служит заполнение буфера цветом). Для 3О-приложения аналогичную функцию выполнял метод Clear.

Затем каждый спрайт коллекции отображается во вторичный буфер, после чего мы можем обновить экран. Для этого вызывается метод Flip, копирующий данные из вторичного буфера в первичный.

При запуске приложения мы должны увидеть результат, схожий с выполнением приложения Direct3D.

ИСПОЛЬЗУЕМЫЕ ИСКЛЮЧЕНИЯ

В данном приложении могут произойти два вида исключительных ситуаций. Первая (WasStillDrawingException) случится при попытке копирования первичной поверхности или рисунка во вторичный буфер в момент времени, когда система еще не завершила предыдущую операцию draw. В Direct3D этот сценарий будет попросту проигнорирован, если вы используете флажок SwapEffect.Discard.
Вторая возможная ситуация — InvalidRectangleException. В Direct3D, если спрайты вышли за пределы экрана, это не так принципиально, поскольку устройство работает в мировых координатах, которые являются, по сути, бесконечными. В DirectDraw это является важным, так как мы не сможем выполнять процедуру рендеринга спрайта, вышедшего за пределы экрана.

Анимация спрайтов

Возьмем за исходный вариант код из предыдущей главы и перепишем его. Класс Sprite подвергнется некоторым изменениям, но некоторые константы мы можем оставить прежними:

```
private const int NumberSpritesRow = 6;
private const int NumberSpritesCol = 5;
private const int SpriteSizeWidth = 50;
private const int SpriteSizeHeight = 45;
```

Исходный текст программы, включенной в CD диск, использует файл sprites.bmp.

ИСПОЛЬЗОВАНИЕ НЕКВАДРАТНЫХ ПОВЕРХНОСТЕЙ

В Direct3D текстуры не должны быть обязательно прямоугольными, но обязаны иметь в качестве длины стороны число, являющееся степенью двойки. Большинство современных плат способны поддерживать нестандартные текстуры, не удовлетворяющие данному ограничению.
В предыдущей главе для анимации мы использовали спрайт, «расширенный» до прямоугольника с соответствующей длиной сторон, для того чтобы такая текстура поддерживалась большинством плат. DirectDraw не использует для создаваемых поверхностей подобные правила, поэтому в нашем примере спрайт имеет размеры, соответствующие изображению.

Мы должны сохранить также положение объекта анимации, которое является комбинацией строки и столбца. Добавьте следующие переменные к классу Sprite:

```
private int column = 0;
private int row = 0;
```

Теперь мы попробуем переписать конструктор для размещения на экране случайного изображения спрайта беспорядочным образом:

```
public GraphicsSprite()
{
    xPosition = rnd.Next(Forml.ScreenWidth-SpriteSizeWidth);
    yPosition = rnd.Next(Forml.ScreenHeight-SpriteSizeHeight);
    xUpdate += (float)rnd.NextDouble ();
    yUpdate += (float)rnd.NextDouble();
    column = rnd.Next(NumberSpritesCol);
    row = rnd.Next(NumberSpritesRow);
    if ((column \ 3) == 0)
        xUpdate *= -1;
    if ((row% 2) == 0)
        yUpdate *= -1;
}
```

Так же понадобиться переписать метод Draw:

```
public void Draw(Surface backBuffer, Surface spriteSurface)
{
   backBuffer.DrawFast(xPosition, yPosition, spriteSurface,
      new Rectangle(column * SpriteSizeWidth, row * SpriteSizeHeight,
   SpriteSizeWidth, SpriteSizeHeight), DrawFastFlags.DoNotWait |
   DrawFastFlags.SourceColorKey);
}
```

Нетрудно заметить, что мы вычисляем исходный полигон, используя текущее изображение и размер спрайта. При прорисовке спрайта мы применили параметр прозрачности и цветовой ключ. Окончательно мы должны модифицировать метод обновления Update с учетом анимации, см. листинг 17.3.

Листинг 17.3. Обновление анимированных спрайтов

```
public void Updated
{
   // Update the current position
   xPosition+= (int)xUpdate;
   yPosition += (int)yUpdate;
```

```
// See if we've gone beyond the screen
if (xPosition > (Forml.ScreenWidth - SpriteSizeWidth))
{
   xUpdate *= -1;
}
if (yPosition > (Forml.ScreenHeight - SpriteSizeHeight))
{
   yUpdate *= -1;
}
// See if we're too high or too the left
if (xPosition < 0)
{
   xUpdate *= -1;
}
if (yPosition < 0)
{
   yUpdate *= -1;
}
// Now update the column
column++;
if (column >= NumberSpritesCol)
{
   row++;
   column = 0;
}
if (row >= NumberSpritesRow)
{
   row = 0;
}
```

После процедуры обновления основная программа должна выдать несколько ошибок при трансляции. Чтобы избежать этого, перепишите код создания спрайта:

```
// Add a few sprites
for(int i = 0; i < 100; i++)
   ar.Add(new GraphicsSprite()) ;
```

Последние два параметра, которые необходимо изменить, находятся в коде рендеринга, при этом необходимо переслать параметры размера спрайта в метод Sprite. Замените этот метод на следующий:

```
protected override void OnPaint(System.Windows.Forms.PaintEventArgs e)
{
   DirectXException.IgnoreExceptions ();
   foreach(GraphicsSprite gs in ar)
```

```
        gs.Updated;
    backBuffer.ColorFill(0);
    foreach(GraphicsSprite gs in ar)
        gs.Draw(backBuffer, sprite);
    primary.Flip(backBuffer, FlipFlags.DoNotWait);
    this.Invalidated;
}
```

Таким образом, мы видим, что различия между Direct3D и DirectDraw не столь значительные, как можно было предположить вначале.

Краткие выводы

В данной главе мы рассмотрели следующие вопросы.

- Использование полноэкранного режима.
- Рендеринг спрайтов.
- Рендеринг анимированных спрайтов.

В следующей главе мы рассмотрим добавление сетевых возможностей, в частности, работу одноранговых сетей.

ЧАСТЬ VI

ДОБАВЛЕНИЕ СЕТЕВЫХ ВОЗМОЖНОСТЕЙ

Глава 18. Организация сети с равноправными узлами с помощью DirectPlay

Глава 19. Создание сессии Client/Server

Глава 20. Особенности более совершенного использования сетей

Глава 21. Методы достижения максимального быстродействия

Глава 18. Организация сети с равноправными узлами с помощью DirectPlay

В этой главе мы рассмотрим работу DirectPlay для сетей с архитектурой "peer-to-peen", включая.

Адреса DirectPlay.

- Равноправные объекты.
- Модель события DirectPlay.

Пересылка данных другим участникам сети.

Адреса DirectPlay

Весьма заманчивым может показаться участие в игре одновременно нескольких игроков. На сегодняшний день имеется целый класс весьма популярных игр, которые вообще не имеют варианта для одного игрока.

Управляемый DirectX включает DirectPlay API, который может использоваться для осуществления сетевых возможностей. Прежде чем начать писать код, необходимо проверить добавление соответствующих директив и ссылок для DirectPlay, в частности, ссылки на Microsoft.DirectX.DirectPlay. Мы полагаем, что все это уже сделано, и не будем останавливаться на этом еще раз.

Если у вас имеется компьютер, подсоединенный к локальной сети или сети Internet, это означает, что он уже имеет свой адрес протокола TCP/IP, который отличает его от других машин в сети.

Мы должны, используя DirectPlay, присвоить каждому компьютеру свой адрес. Процедура создания индивидуального адреса в DirectPlay выглядит достаточно просто:

```
Address address = new Address();
address.ServiceProvider = Address.ServiceProviderTcpIp;
```

Здесь мы указываем, что хотим использовать для DirectPlay протокол TCP/IP, который является на сегодняшний день наиболее распространенным. Ниже приводятся еще несколько протоколов и соединений, доступных для DirectPlay.

- Протокол TCP/IP.

Протокол IPX.

- Технология BlueTooth.
- Соединения с использованием последовательных портов.

Соединения модемов напрямую.

В дальнейшем, при написании приложений DirectPlay мы будем иметь дело только с TCP/IP протоколом. Но это вовсе не означает, что мы не можем использовать другие протоколы, поэтому записанный код может быть применим и к другим перечисленным выше протоколам и соединениям.

ИСПОЛЬЗОВАНИЕ «URL» СТРОКИ В КАЧЕСТВЕ АДРЕСА

Адреса могут быть также определены в форме указателя URL, как и все Web-адреса, например, http://www.mycompany.com. Первая секция URL указывает на тип протокола (в данном случае протокол http). Web-страницы, которые мы обычно видим, имеют формат языка HTML. Ниже приводится пример использования указателя URL для задания адреса в DirectPlay:

```
x-directplay:/provider=%7BEBFE7BA0-628D-HD2-
    AE0F-006097B014H%7D;hostname=
    www.mygameserver.com;port=9798
```

Обратите внимание на тип этого URL— x-directplay.
Остальная часть адреса определяет идентификатор провайдера, имя хоста и порт.

ИСПОЛЬЗОВАНИЕ TCP/IP АДРЕСА

Имеется четыре конструктора для адресного объекта, большинство из которых имеет дело с TCP/IP протоколом. Используемый нами ранее и не имеющий параметров конструктор не устанавливает протоколы автоматически. Другие три конструктора задействуют протокол TCP/IP автоматически, рассмотрим их подробнее:

```
public Address ( System.String hostname , System.Int32 port )
public Address ( System.Net.IPAddress address )
public Address ( System.Net.IPEndPoint address )
```

Каждый из этих конструкторов выполняет одну и ту же операцию, только с различными наборами данных. Первый создает адрес TCP/IP, устанавливает имя хоста и соответствующий порт. Имя хоста может быть именем компьютера, адресом Internet (например, www.mycompany.com) или IP-адресом (например, 192.168.2.1). Два других варианта могут принимать существующие адреса TCP/IP и конвертировать их в соответствующий адрес DirectPlay.
До сих пор мы не использовали такие термины, как «имя хоста» и «порт». Попробуем с помощью адресного объекта написать проце-

дуру добавления этих компонентов к нашему адресу. Каждый компонент имеет название (значение в формате string) и ассоциированные данные (либо строчные, либо в виде массива). Да и сам класс Address содержит стандартные ключевые имена для создаваемых адресов. Запишем код, позволяющий вручную присвоить имя хоста и порт существующему адресу:

```
Address address = new Address));
address.ServiceProvider  =  Address.ServiceProviderTcpIp;
address.AddComponent(Address.KeyHostname,  "www.mygameserver.com");
address.AddComponent(Address.KeyPort,   9798);
```

В действительности, конструкторы класса Address, имеющие входные параметры, выполняют эти действия достаточно точно.

Помимо этого существует несколько заданных по умолчанию ключевых имен, которые могут использоваться в различных случаях (например, KeyPhoneNumber для прямых соединений модем-модем), кроме того, пользователь может создавать и свои имена.

Компонент «hostname» особенно важен, когда мы связываемся с другим компьютером. При этом DirectPlay достаточно гибок, чтобы распознать имя хоста на вашей машине, и в этом случае нет необходимости использовать указанные ключи для локальных адресов.

Создание P2P-соединения

Теперь мы можем перейти к созданию сети. Следует отметить, что DirectPlay поддерживает системы и с архитектурой «peer-to-peen> (равный с равным), и с архитектурой «клиент-сервер». В этой главе мы обсудим соединения с архитектурой «peer-to-peer», которые в дальнейшем будем называть пиринговыми или P2P-соединениями.

Архитектура P2P подразумевает, что каждый участник соединения связан непосредственно с каждым другим участником соединения. Для небольшого числа узлов этот метод работает достаточно надежно. Имеется множество популярных на сегодняшний день примеров применения P2P-сетей. Многие из систем совместного использования файлов (Kazaa, Napster и т. д.) представляют собой именно P2P-сети.

Прежде чем начать работу в DirectPlay с P2P-соединением, необходимо уточнить несколько вопросов: например, выбрать тип соединения и определить, будет'ли наша машина поддерживать работу сети или присоединится к уже существующему сеансу.

Основным классом, который мы будем использовать для нашего соединения, будет класс Peer. Для начала объявим переменные соединения и адреса для этого класса:

```
private Peer connection = null;
private Address deviceAddress = null;
```

Далее создадим в верхнем правом углу наше формы две кнопки. Первая кнопка «Host» будет использоваться для организации собственной сессии, а вторая «Connect» — для подключения к существующему сеансу. Добавьте там же третью кнопку «Send Data», которая будет управлять передачей данных в сети (до соединения эта кнопка должна быть заблокирована). Затем в нижней части экрана создадим экран индикации состояния сети и приступим к работе с сетью.

Вначале мы запишем функцию инициализации объекта Peer и установим любое заданное по умолчанию состояние, см. листинг 18.1.

Листинг 18.1. Инициализация объекта Peer.

```
private void InitializeDirectPlay(bool host)
{
   // Create our peer object
   connection = new Peer() ;
   // Check to see if we can create a TCP/IP connection

   if  (!IsServiceProviderValid(Address.ServiceProviderTcpIp))
   {
      // Nope, can't, quit this application
      MessageBox.Show("Could not create a TCP/IP service provider.", "Exiting",
         MessageBoxButtons.OK, MessageBoxIcon.Information) ;
      this.Close();
   }
   // Create a new address for our local machine
   deviceAddress = new Address();
   deviceAddress.ServiceProvider = Address.ServiceProviderTcpIp;
   }
```

При создании объекта Peer мы использовали конструктор, задающий свойства объекта по умолчанию. Существует и другой вариант, который имеет в качестве параметров флажки инициализации, приведенные в таблице 18.1.

Таблица. 18.1. Флажки объекта

Флажок	Описание
DisableParamaterValidation	Данный флажок отключает проверку параметров этого соединения, позволяя получить выигрыш в быстродействии

Флажок	Описание
DisableLinkTuning	Отключает возможность настройки скорости передачи данных в DirectPlay, автоматически устанавливается максимальная скорость
HintLanSession	Открывает расширенное окно для сетевых игр
None	Позволяет установить все опции по умолчанию

Эти опции могут быть весьма полезны, однако сейчас они нам не понадобятся, поэтому мы будем использовать для этого приложения конструктор, устанавливающий свойства по умолчанию. Так как для работы приложения нам потребуется служба поддержки протокола TCP/IP, мы должны проверить ее наличие в нашей системе. Для этого можно использовать метод IsServiceProviderValid:

```
private bool IsServiceProviderValid(Guid provider)
{
    // Ask DirectPlay for the service provider list
    ServiceProviderInformation[] providers =
        connection.GetServiceProviders(true);
    // For each service provider in the returned list...
    foreach (ServiceProviderInformation info in providers)
    {
        // Compare the current provider against the passed provider
        if (info.Guid == provider)
            return true;
    }
    // Not found
    return false;
}
```

Эта процедура определяет все имеющиеся в нашей системе службы поддержки протоколов и возвращает информацию обо всех обнаруженных в системе службах, в том числе и тех, которые на данный момент недоступны (при значении входного параметра функции равного «true»). Если в возвращаемом массиве имеется идентификатор GUID, это говорит о том, что служба поддержки обнаружена, и мы можем приступить к созданию адреса. В случае если протокол TCP/IP не поддерживается, приложение завершит работу. Мы также должны предусмотреть возможность освобождения объекта Peer после завершения соединения, для этого имеется перегрузка по умолчанию Dispose:

```
if (connection != null)
    connection.Dispose();
```

Теперь мы должны записать код для каждого сценария.

Начало сеанса

Итак, в нашей форме имеются две кнопки, соответственно, для сеанса Host и для сеанса Connect. Мы должны исключить возможность нажатия одновременно двух кнопок или нажатия одной из них во время сеанса. Соответствующие обработчики событий приведены ниже:

```
private void button1_Click(object sender, System.EventArgs e)
{
   button1.Enabled = false;
   button2.Enabled = false;
   InitializeDirectPlay(true);
}
private void button2_Click(object sender, System.EventArgs e)
{
   button1.Enabled = false;
   button2.Enabled = false;
   InitializeDirectPlay(false);
}
```

Как вы можете видеть, разница между ними состоит только в конечном параметре, передающемся в метод InitializeDirectPlay (значение параметра «true» для установки компьютера главным, другое значение — «false» — для присоединяемой машины). Мы могли бы упростить данную процедуру, сделав ее общей для обеих кнопок:

```
private void button_Click(object sender, System.EventArgs e)
{
   button1.Enabled = false;
   button2.Enabled = false;
   InitializeDirectPlay((sender == button1));
}
```

Результат выполнения этой процедуры такой же, но при этом мы избегаем дублирования кода. Если нажатой окажется первая кнопка, метод InitializeDirectPlay будет вызываться с входным параметром, имеющим значением «true», в противном случае, со значением «false».

Теперь необходимо написать метод, распознающий и сообщающий приложению о выбранном сеансе подключения. DirectPlay включает структуру описания приложения, которая содержит всю необходимую информацию. Нижеприведенный код для этой структуры необходимо добавить в конец метода InitializeDirectPlay:

```
// Set up an application description
ApplicationDescription desc = new ApplicationDescription();
desc.SessionName = "MDXBookPeerSession";
```

```
desc.GuidApplication = new Guid(41039, 1702,1503,178, 101,
    32, 13, 121, 230, 109, 59);
```

Наиболее важный параметр этой структуры — GuidApplication. Идентификатор GUID однозначно определяет наше приложение, и таким образом, все примеры данного приложения должны его использовать. Другой параметр — имя сеанса (SessionName) — необходим при выполнении нескольких сеансов связи. В таблице 18.2 приведены остальные параметры структуры ApplicationDescription.

Таблица 18.2. Параметры структуры ApplicationDescription

Параметр	**Описание**
GuidApplication	Уникальный идентификатор приложения
GuidInstance	Уникальный идентификатор соединения, который генерируется DirectPlay. Идентифицирует отдельные экземпляры приложения
MaxPlayers	Максимальное число пользователей в данном соединении. Нулевое значение (по умолчанию) устанавливает неограниченное число игроков
CurrentPlayers	Число подсоединенных на данный момент пользователей
Flags	Определяют поведение соединения, могут использоваться по отдельности или в различных комбинациях: • ClientServer • FastSigned • FullSigned • MigrateHost • NoDpnServer • NoEnumerations • RequirePassword
SessionName	Имя соединения, определенное пользователем
Password	Пароль для установления соединения. Это значение должно иметь пустой указатель (null) в случае, если флажок RequirePassword не был установлен

Теперь, имея структуру описания, мы можем приступить к созданию соединения. Добавьте следующий код в конец метода инициализации:

```
if (host)
{
    try
    {
        // Host a new session
```

```
        connection.Host(desc, deviceAddress);
        // We can assume after the host call succeeds we are hosting
        AddText("Currently Hosting a session.");
        EnableSendDataButton(true);
    }
    catch
    {
        AddText("Hosting this session has failed.");
    }
}
```

Как вы можете видеть, вызов режима Host оказался в достаточной мере «простым», входными параметрами этой функции являются описание соединения и локальный адрес. Здесь мы использовали самый простой из восьми существующих вариантов загрузки. Остальные также используют в качестве входных параметров указанное описание и один или более адресных объектов.

В качестве параметра может использоваться объект, содержащий информацию об игроке (мы кратко опишем это чуть позже), или флажок, определяющий поведение соединения, например, позволяющий DirectPlay открыть диалоговое окно для получения недостающей информации об адресах.

ПАРОЛИ НЕ ЗАШИФРОВАНЫ

Будьте внимательны при передаче паролей. Если вы хотите защитить пароли, необходимо перед пересылкой зашифровать их вручную.

Итак, далее мы помещаем вызов процедуры Host в блок кода, позволяющий отследить событие, когда метод по каким-либо причинам не выполняется. Мы могли бы отслеживать только те исключительные ситуации, которые входят в класс DirectPlayExcetion, но для нашего простого примера мы попросту рассмотрим все подобные случаи. Но сначала создадим процедуры, позволяющие модифицировать пользовательский интерфейс:

```
private void AddText(string text)
{
    labell.Text += (text + "\r\n");
}

private void EnableSendDataButton(bool enable)
{
    button3.Enabled = enable;
```

Нет необходимости останавливаться на этих методах подробно, поскольку они весьма очевидны. Запуск приложения и нажатие кнопки «Host» приведет к установке P2P-соединения, отобразит на экране запись о текущем состоянии и сделает доступным использование третьей кнопки «Send Data». Теперь нам необходим еще один узел для установки сеанса соединения. Добавим следующую процедуру в конец метода инициализации:

```
else
{
   try
   {
      connection.FindHosts(desc, null, deviceAddress, null,
         0, 0, 0, FindHostsFlags.OkToQueryForAddressing);
      AddText("Looking for sessions.");
   }
   catch
   {
      AddText("Enumeration of sessions has failed.");
   }
}
```

Мы снова включаем процедуру в блок try/catch, чтобы отследить возможные ошибки, а также обновляем наше сообщение пользователю. Однако, здесь вместо метода Connect мы вызываем метод FindHosts. В некоторых случаях возможно непосредственное присоединение к имеющемуся сеансу, но в большинстве случаев вначале необходимо найти существующие хосты.

Аналогично методу Host, метод FindHosts принимает описание соединения и локальный адрес устройства. Вторым параметром является адрес главного компьютера (пока мы его не знаем, поэтому используем пустой указатель). Четвертый параметр этого метода — любые данные о приложении, которые мы хотим передать на сервер.

Следующие три параметра определяют режим работы хостов: количество попыток передачи пакетов, ожидание перед следующей передачей данных и время таймаута. Задавая нулевые значения этих параметров, мы определяем установку для них режима по умолчанию. Последний параметр может иметь одно или больше значений из списка FindHostsFlags, который приведен в таблице 18.3.

Таблица 18.3. Значения и флажки структуры FindHostFlags

Параметр	Описание
None	Устанавливает все значения по умолчанию

Параметр	Описание
NoBroadcastFallback	Отключает режим ретрансляции данных на сервере. Поддержку возможности передачи данных службой поддержки можно проверить с помощью метода GetSpCaps класса Peer
OkToQueryForAddressing	Позволяет устройству DirectPlay отображать диалоговые окна для уточнения текущей информации
Sync	По умолчанию выполнение процедуры заканчивается немедленно. При использовании данного флажка процедура будет функционировать до тех пор, пока не завершатся все попытки соединения

Запустив теперь приложение, обратите внимание, что нажатие на кнопку «Connect» вызовет диалоговое окно, запрашивающее адрес удаленной машины. Вы можете оставить это поле пустым, чтобы отследить состояние всей сети. Вы можете также определить имя или IP-адрес искомой удаленной машины. Следует отметить, что метод FindHosts не возвращает никаких сведений относительно найденных хостов.

Использование модели событий

Большинство действий в DirectPlay выполняются асинхронно. Результат выполнения процедуры возвращается немедленно, и сразу после этого DirectPlay начинает отработку поставленных задач. На данном этапе нам понадобится обработчик ситуации, когда DirectPlay закончил выполнение операции или выдал сообщение о состоянии приложения.

В нашем предыдущем вызове метода FindHosts выполнялся поиск всех возможных соединений. Допустим, хост был найден. Поскольку нам необходимо об этом знать, мы должны зафиксировать данное событие (FindHostResponsee). Добавьте следующую строку сразу после создания объекта Peer в методе InitializeDirectPlay:

```
connection.FindHostResponse += new
    FindHostResponseEventHandler(OnFindHost);
```

Мы также должны создать обработчик этого события, см. листинг 18.2.

Листинг 18.2. Обработчик события обнаружения «Found Host».

```
private void OnFindHost(object sender, FindHostResponseEventArgs e)
 {
    lock(this)
    {
```

```
            //Do nothing if we're connected already
            if  (connected)
               return;
            connected = true;
            string foundSession = string.Format
               ("Found session ((0}), trying to connect.",
               e.Message.ApplicationDescription.SessionName);
            this.BeginInvoke(new AddTextCallback(AddText),
               new object[] { foundSession });
            // Connect to the first one
            ((Peer)sender).Connect(e.Message.ApplicationDescription,
               e.Message.AddressSender,
            e.Message.AddressDevice, null, ConnectFlags.OkToQueryForAddressing);
        }
    }
```

Как вы можете видеть, данный обработчик возвращает объект FindHostsReponseEventArgs, в котором содержится информация о найденном хосте. Вполне возможно, что могут быть одновременно найдены два хоста или соединения. Наличие двух процессов, оперирующих одним набором данных, может привести к возникновению больших проблем, и нам важно избежать данной ситуации. Мы будем использовать для блокировки одного из потоков ключевое слово языка C# lock (которое по сути является аналогом вызова комбинации Monitor.Enter Monitor.Exit в блоке try/finally). Таким образом, при использовании первого потока второй поток блокируется.

В итоге мы сможем присоединиться только к одному хосту, поэтому мы должны объявить логическую переменную, определяющую наличие уже установленного соединения. Если соединение существует, мы можем вернуться из процедуры, в противном случае переменной присваивается значение «false», чтобы поиск соединения продолжался дальше.

Теперь нам необходимо сообщить пользователю, что сеанс обнаружен, и мы готовы начать соединение. Для этого в программе используется метод BeginInvoke, вызывающий так называемый делегат с набором параметров, определенным управляющим узлом. Данный метод работает асинхронно (классический вариант для DirectPlay). Вызвав BeginInvoke, мы можем продолжать работу нашего приложения, а делегат будет выполнятся параллельно.

Имеется также метод Invoke, который принимает те же самые параметры, но выполняется синхронно. Определение для делегата имеет вид:

```
private delegate void AddTextCallback(string text);
```

Таким образом, мы создали делегат для вызова уже существующего метода AddText. Данные действия позволяют не блокировать поток при

отображении диалогового окна, как можно быстрее получить необходимые нам данные и завершить процедуру.

Наконец, мы готовы начать соединение (вернемся к рассмотрению листинга 18.2). Вызов подключения Connect имеет четыре модели загрузки, и, как обычно, мы используем самую простую из них. Обратите внимание, что для передачи в процедуру Connect мы используем параметры описания, принятые от обработчика FindHostsEventArgs.

Последние два пересылаемых нами параметра определены нашим приложением. Первый из них — любые определяемые пользователем данные, которые мы хотим передать в вызов Connect. Второй позволяет DirectPlay сделать запрос на получение адресной информации, хотя в этом нет особой необходимости, поскольку мы уже имеем ее. Для того чтобы обеспечить синхронное выполнение, необходимо задействовать флажок Sync, напомним, что по умолчанию установлен обратный вариант.

Другие модели метода Connect содержат один или несколько параметров, которые мы не используем в нашем примере. Одним из выходных параметров может являться асинхронный обработчик вызова (в DirectPlay все асинхронные обращения могут возвращать подобный параметр), который при необходимости можно использовать, чтобы отменить операцию с помощью функции CancelAsyncOperation. Но здесь нам это не понадобится. Другие параметры являются определяемыми пользователем контекстными переменными, которые могут использоваться для записи специфических данных.

АДРЕСНАЯ ИНФОРМАЦИЯ

При попытке присоединиться к полноэкранному сетевому приложению желательно, чтобы любая информация о возможных сетевых соединениях появлялась внутри пользовательского интерфейса игры, а не в отдельном диалоговом окне.

При разработке сетевых уровней для полноэкранного приложения вам вряд ли захочется или, более того, понадобится пересылать флажки OkToQueryForAddressing в методы Connect, Host или FindHost, поскольку при этом DirectPlay отобразит свое собственное диалоговое окно, которое, скорее всего, не будет соответствовать пользовательскому интерфейсу вашего приложения.

Если вы используете указанные методы без этого флажка, и во время их выполнения возникает исключительная ситуация, вполне вероятно, что был пропущен какой-либо компонент адреса. Для варианта с протоколом TCP/IP это могло бы быть имя хоста (или сервера) или используемый порт. Предварительно необходимо убедиться в наличии необходимых адресов и их компонентов.

Работа в сети

Теперь мы могли бы записать алгоритм обработчика, отслеживающего завершение метода Connect. Для начала добавьте следующие строки к методу InitializeDirectPlay:

```
connection.ConnectComplete += new
   ConnectCompleteEventHandler(OnConnectComplete);
```

Здесь же мы можем добавить и сам обработчик:

```
private void OnConnectComplete(object sender, ConnectCompleteEventArgs e)
(
   // Check to see if we connected properly
   if (e.Message.ResultCode == ResultCode.Success)
   (
      this.BeginInvoke(new AddTextCallback(AddText),
         new object[] ( "Connect Success." });
      connected = true;
      this.BeginInvoke(new EnableCallback(EnableSendDataButton),
         new object[] ( true ( );

   else
   (
      this.BeginInvoke(new AddTextCallback(AddText), new object[] (
         string.Format("Connect Failure: (0|", e.Message.ResultCode) lb-
      connected = false;
      this.BeginInvoke(new EnableCallback(EnableSendDataButton),
         new object[] ( false ( );

)
```

Таким образом, мы получаем информацию об успешном или неудачном соединении. В случае удачного соединения мы можем задействовать кнопку для передачи данных. В противном случае мы наоборот блокируем кнопку для передачи данных и присваиваем переменной наличия соединения значение «false», и тогда обработчик FindHosts продолжит отслеживать другие возможные соединения. Обратите внимание, что нам необходим делегат, управляющий режимом использования кнопки для передачи данных. Мы можем объявить его следующим образом:

```
private delegate void EnableCallback(bool enable);
```

Теперь попробуем запустить два примера приложения и соединить их друг с другом. В первом примере нажимаем кнопку «Host», при этом должно появиться сообщение о том^что это приложение будет являться хос-

том соединения. Далее нажимаем кнопку «Connect» во втором примере, и после запроса об удаленном хосте мы должны увидеть результат соединения.

Попытаемся переслать некоторые данные с помощью этого соединения. Для этого необходимо добавить соответствующий обработчик для кнопки передачи данных:

```
private void button3_Click(object sender, System.EventArgs e)
{
    NetworkPacket packet = new NetworkPacket();
    packet.Write(byte.MaxValue);
    connection.SendTo((int)PlayerID.AllPlayers, packet,
        0, SendFlags.Guaranteed);
}
```

При пересылке данных по сети, как правило, используется сетевой пакет, в который мы можем записывать любую информацию определенного формата, за исключением ссылок на данный метод. Для этого приложения нам не важен тип данных, поэтому для простоты мы будем передавать число в виде отдельного байта.

Как мы видим, вызов метода Send весьма прост. Первый параметр PlayerID — идентификационный номер игрока (или группы игроков, это мы обсудим позже), которому мы хотим послать данные. В данном примере мы посылаем данные всем игрокам. Вторым параметром является собственно сам сетевой пакет. Третий параметр — значение таймаута для передачи этого пакета. Нулевое значение этого параметра приведет к продолжению попыток передачи пакета до того, как соединение прервется. И последний параметр в этом вызове — один или несколько флажков структуры SendFlags. Примеры этих флажков и их описание приведены в таблице 18.4.

Таблица. 18.4. Флажки структуры SendFlags

Параметр	Описание
Sync	Устанавливает синхронные операции вместо заданных по умолчанию асинхронных
NoCopy	Данный флажок выполняется только для перегрузок, принимающих в качестве сетевого пакета структуру GCHANDLE, позволяет DirectPlay использовать данные в GCHANDLE непосредственно, без создания отдельной копии. Данный флажок не может использоваться с флажком NoComplete
NoComplete	Использование этого флажка позволяет не отслеживать событие SendComplete. Не используется с флажками NoCopy или Guaranteed

Параметр	Описание
CompleteOnProcess	Использование этого флажка позволяет осуществить проверку соединения и отследить событие передачи данных SendComplete. Используется с флажком Guaranteed
Guaranteed	Посылает сообщение для проверки наличия соединения
PriorityHigh	Посылает сообщение с высоким приоритетом. Не может использоваться совместно с флажком PriorityLow
PriorityLow	Посылает сообщение с низким приоритетом. Не может использоваться совместно с флажком PriorityHigh
Nonsequential	По умолчанию DirectPlay определяет тот же самый порядок приема данных, что и при передаче. Если сообщения достигают компьютера в другом порядке, они будут буферизированы и переупорядочены. При установке данного флажка данные не переупорядочиваются
NoLoopBack	Задание этого флажка позволяет не отслеживать событие приема данных при пересылке их другому игроку или группе, в которой вы находитесь
Coalesce	Использование этого флажка позволяет DirectPlay объединять пересылаемые пакеты

Теперь, когда мы, нажав соответствующую кнопку, отослали данные, нам понадобится процедура, отслеживающая момент приема данных:

```
connection.Receive += new
   ReceiveEventHandler(OnDataReceive);
Here is the code for the actual event handler:
private void OnDataReceive(object sender, ReceiveEventArgs e)
{
   // We received some data, update our UI
   string newtext = string.Format
       ("Received message from DPlay Userld: 0x{0}",
       e.Message.SenderlD.ToString("x"));
   this.Beginlnvoke(new AddTextCallback(AddText),
       new object[] { newtext });
}
```

Метод достаточно прост, мы регистрируем событие приема данных (ReceiveEventArgs) и уведомляем об этом пользователя.

Попробуем запустить два примера приложения (не важно, на одной или на двух машинах) и соединить их друг с другом. Как и раньше, у нас име-

ется и главное, и подключаемое приложения. Нажатие кнопки «Send Data» вызовет появление сообщений пользователю на обеих машинах. Это произойдет автоматически, поскольку мы не использовали флажок NoLoopBack при вызове метода SendTo, и данные были переданы на всем участникам.

Теперь, не завершая приложения, отключим соединение на главном компьютере и нажмем кнопку «Send Data» на втором экземпляре. Очевидно, что возникла ситуация, которая не может быть обработана, и приложение «рухнуло». Рассмотрим этот вариант подробнее.

Обработка потерянных сессий

Когда главный компьютер завершает сеанс работы, соединение разрывается. Необходимо написать процедуру, позволяющую отследить данное событие и оповестить всех текущих пользователей о разрыве соединения:

```
connection.SessionTerminated += new
   SessionTerminatedEventHandler(OnSessionTerminate);
```

Разрыв связи (событие Session.Terminated) возможен по различным причинам, например: отсоединение кабеля, выход из сеанса и т. д. Код для обработчика подобной ситуации имеет следующий вид:

```
private void OnSessionTerminate(object sender, SessionTerminatedEventArgs e)
{
   // The session was terminated, close our peer object, re-enable our buttons
   this.BeginInvoke(new DisconnectCallback(OnDisconnect), null);
}
```

Итак, у нас имеются новый делегат и новый метод, которые должны быть вначале объявлены:

```
private delegate void DisconnectCallback0;
private void OnDisconnect()
{
   // Re-enable the UI
   button1.Enabled = true;
   button2.Enabled = true;
   button3.Enabled = false;
   // Notify the user
   AddText("The host has quit, you have been disconnected.");
   // Dispose of our connection, and set it to null
   connection.Disposed;
   connection = null;
}
```

Теперь понятно, почему мы выбрали такой подход. После разрыва соединения мы сначала устанавливаем состояние кнопок в исходное положение (по умолчанию), чтобы подготовиться к новому сеансу подключения. Затем обновляется пользовательский интерфейс и, наконец, поскольку старое соединение уже не имеет силы, устанавливается нулевое значение соединения.

ОБРАБОТКА ПЕРЕДАЧИ ХОСТА

Весьма заманчивой кажется передача управления после выхода хоста из сеанса любому из имеющихся узлов сети. В DirectPlay имеется возможность передачи управления сеансом, которая автоматически выберет новый хост из оставшихся машин. В разделе конструирования нашей формы между кнопками «Connect» и «Send" Data» добавьте кнопку «Host Migration».

Теперь, чтобы «оживить» данную опцию, мы должны переписать несколько процедур в нашем приложении. В начале мы должны определить соответствующий флажок передачи перед вызовом Host:

```
// Should we allow host migration?
desc.Flags = checkBoxl.Checked ? SessionFlags.MigrateHost : 0;
```

Как видите, если возможность этого действия проверена, включается флажок MigrateHost. Также мы должны включить управление этой кнопкой в методе OnDisconnect и в процедуре обработки нажатия кнопки (по аналогии с кнопками «Host» и «Connect»).

Наконец, было бы неплохо обновить пользовательский интерфейс при выходе главного компьютера из соединения и передаче функций хоста следующей машине. Добавим необходимый для этого код:

```
connection.HostMigrated += new
   HostMigratedEventHandler(OnHostMigrate);
```

Обработчик этого события имеет следующий вид:

```
private void OnHostMigrate(object sender, HostMigratedEventArgs e)
(
   // The host has been migrated to a new peer
   this.Beginlnvoke(new AddTextCallback(AddText),
      new object[] ( "The host was migrated." });
}
```

Если бы наше приложение имело специфические опции хоста, мы могли бы задействовать их в этом методе, проверив значение параметра NewHostId в объекте HostMigratedEventArgs.

Краткие выводы

В этой главе мы рассмотрели принципы использования DirectPlay для P2P-сетей, включая.

- Адреса DirectPlay.
- Равноправные объекты.
- Модель события DirectPlay.
- Пересылка данных другим участникам сети.
- Передача функций хоста.

В следующей главе мы рассмотрим работу в режиме мультиигрока с использованием клиент-серверной топологии.

Глава 19. Создание сессии Client/Server

P2P-сети работают достаточно хорошо, когда мы имеем дело с небольшим числом пользователей. Однако, когда речь заходит о сотнях, а то и тысячах игроков или пользователей, необходимо наличие выделенного сервера. При работе P2P-сети все передаваемые данные могли быть продублированы каждому подключенному объекту. Для небольших групп эта операция вполне приемлема, но если в сети участвуют 10000 машин, это может надолго «подвесить» выполнение операций.

В DirectPlay имеется два других сетевых класса, подобных рассмотренному нами объекту Peer: это класс Client и класс Server, соответственно клиент и сервер. В этой главе мы с помощью DirectPlay создадим приложение клиент-сервер и рассмотрим следующие вопросы.

- Создание выделенных серверов.
- Соединение с этими серверами, используя клиентский интерфейс.
- Отслеживание подключения и отключения игрока.
- Передача игровых данных.

Создание выделенного сервера

При создании приложения клиент-сервер, в первую очередь создается сервер. Большинство серверных приложений используют порты Internet. Например, Web-серверы практически всегда используют 80-й порт, а серверы FTP — 21-й, причем с открытым портом одновременно может работать только одно приложение. Наша задача — определить индивидуальный порт, с которым будет работать наше приложение. Первая тысяча номеров считается зарезервированной, хотя мы можем свободно использовать любой доступный порт.

Для нашего приложения мы выберем порт 9798. Мы пока не будем использовать для нашего сервера какие-либо опции, только определим его пользовательский интерфейс и простейшие действия.

Как обычно, мы должны создать нашу форму, добавить необходимые ссылки и директивы для DirectPlay и описать диалоговое окно. Подобно тому, как мы объявляли объект Peer, объявим объект Server:

```
private Server connection = null;
```

В листинге 19.1 приведена процедура инициализации объекта Server. Здесь нет смысла определять логическую переменную для проверки наличия соединения поскольку мы создаем только сервер без каких-либо соединений.

Лисияг 19.1. Инициализация Сервера.

```
public void InitializeServerO
 {
    // Create our server object
    connection = new Server!);
    // Check to see if we can create a TCP/IP connection
    if  (!IsServiceProviderValid(Address.ServiceProviderTcpIp))
    {
       // Nope, can't, quit this application
       MessageBox.Show("Could not create a TCP/IP service provider.", "Exiting",
          MessageBoxButtons.OK, MessageBoxIcon.Information);
       this.Close();
    }
    // Create a new address for our local machine
    Address deviceAddress = new Address();
    deviceAddress.ServiceProvider = Address.ServiceProviderTcpIp;
    deviceAddress.AddComponent(Address.KeyPort, SharedCode.DataPort);
    // Set up an application description
    ApplicationDescription desc = new ApplicationDescription();
    desc.SessionName = "MDX Book Server Session";
    desc.GuidApplication = SharedCode.ApplicationGuid;
    desc.Flags = SessionFlags.ClientServer | SessionFlags.NoDpnServer;
    try
    {
       // Host a new session
       connection.Host(desc, deviceAddress) ;
       // We can assume after the host call succeeds we are hosting
       AddText("Currently Hosting a session.");
    }
    catch
    {
       AddText("Hosting this session has failed.");
    }
 }
```

Данный метод подобен методу инициализации объекта Peer. Методы IsServiceProviderValid и AddText были объявлены в предыдущей главе, поэтому здесь мы не станем на них останавливаться. Глядя на метод инициализации сервера, можно обнаружить несколько важных отличий его создания от создания P2P-сети.

Здесь мы добавили к адресу компонент порта (константа SharedCode.DataPort). Поскольку указанная константа требуется при создании и сервера, и клиента, необходимо создать отдельный фрагмент кода ShareCode, который будет использоваться обеими процедурами:

```
public class SharedCode
{
   public static readonly Guid ApplicationGuid = new Guid
      (15741039, 1702,1503,178, 101, 32, 13, 121, 230, 109, 59);
   public const int DataPort = 9798;
}
```

Каждая из двух определенных нами констант понадобится нам в обеих процедурах создания объектов. Для P2P-сети весь выполняемый код формировался в одной программе, и не было необходимости в написании отдельного кода для совместного использования. В нашем случаг мы создаем совместный файл кода ShareCode.

Другим отличием в приведенном методе являются необходимые длл описания приложения флажки. В нашем случае используются флажки ClientServer и NoDpnServer. Первый флажок достаточно очевиден, второй же требует пояснений.

Как уже упоминалось выше, к отдельному порту одновременно может обращаться только одно приложение. Однако, существует поддерживаемый DirectPlay сценарий выполнения на машине нескольких серверных программ, оперирующих с одним и тем же портом.

Для работы в таком режиме в DirectPlay имеется внешнее приложение «dpnsrv.exe», которое запускается при первом создании соединения. Данное приложение привязывается к определенному или заданному по умолчанию порту и пересылает любые клиентские данные на «реальный» хост-компьютер. Иногда использование данного режима весьма полезно, но для нашего приложения в этом нет необходимости, и мы должны использовать флажок NoDpnServer для игнорирования вызова приложения «dpnsrv.exe».

Прежде чем запускать и использовать сервер, мы должны убедиться в корректной работе процедур инициализации и завершения соединения в конце сеанса. Перепишите основной метод следующим образом:

```
static void Main()
{
   using (Forml frm = new Form1())
   {
      frm.Show();
      frm.InitializeServer();
      Application.Run(frm);
   }
}
```

Также необходимо изменить функцию Dispose (освобождение объекта) для нашей формы, добавив соответствующий код:

```
if (connection != null)
    connection.Dispose ();
```

Чтобы почувствовать влияние использования флажка NoDpnServer, нужно откомпилировать и запустить приложение. Мы увидим сообщение о том, что наше соединение установлено и является главным. Теперь, не прерывая сеанс работы первого примера, попробуем запустить приложение еще раз. Мы должны получить сообщение об ошибке и невозможности соединения. Причина кроется в том, что второй экземпляр запускался при обращении к уже использующемуся порту. Теперь удалите флажок, и попробуйте проделать то же самое. Результатом будет одновременная работа двух хостов на нашем сервере.

Теперь было бы желательно вернуть на место удаленный флажок NoDpnServer и продолжить написание программы.

Создание соединения клиент-сервер

Создав выделенный сервер, попробуем теперь создать соединение клиент-сервер. В принципе, на данном этапе можно было бы добавить процедуру создания клиента в программу для сервера, но поскольку мы хотим иметь выделенный сервер, запишем отдельный код для клиентской части. Для этого нам понадобится сконструировать еще одну windows-форму.

Пользовательский интерфейс и свойства диалогового окна клиента будут такими же, как и для сервера. Следует отметить, что для клиентской части необходимо выделить больше места в форме под кнопки основных операций. Также нашему соединению понадобится имя. Для этого создадим в форме окно (аналогично плашке textbox), в котором сможем изменять имя клиента.

Далее мы должны удостовериться, что имеем доступ к общим данным сервера и клиента. При необходимости нужно добавить ссылку на общий файл кода ShareCode, который был создан нами для сервера. При добавлении данной ссылки убедитесь в том, что нажали маленькую стрелку на кнопке «Open», и выбрали опцию Link file (тогда файл кода будет формироваться в отдельной папке, иначе в локальной папке будет создана копия файла, и данные более не будут доступны для совместного использования нашими приложениями).

Выполнив подготовительные действия, мы можем приступить к написанию кода для клиентской части, объявив в начале необходимые локальные переменные:

```
private Client connection = null;
private bool connected = false;
```

Так же как и для P2P-сеанса, для проверки наличия соединения между клиентом и сервером нам понадобится соответствующая переменная.

Как обычно, запишем метод инициализации, включающий в себя возможность поиска имеющихся в сети хостов. Добавьте метод, приведенный в листинге 19.2 к нашему клиентскому приложению.

Листонг 19.2. Инициализация соединения Client.

```
public void InitializeClient()
 {
    // Create our client object
    connection = new Client));
    // Hook the events we want to listen for
    connection.ConnectComplete += new
       ConnectCompleteEventHandler(OnConnectComplete);
    connection.FindHostResponse += new
       FindHostResponseEventHandler(OnFindHost);
    // Check to see if we can create a TCP/IP connection
    if  (!IsServiceProviderValid(Address.ServiceProviderTcpIp))
     {
    // Nope, can't, quit this application
    MessageBox.Show("Could not create a TCP/IP service provider.", "Exiting"
       MessageBoxButtons.OK, MessageBoxIcon.Information);
    this.Close();
     }
    // Create a new address for our local machine
    Address deviceAddress = new Address();
    deviceAddress.ServiceProvider = Address.ServiceProviderTcpIp;
    // Create a new address for our host machine
    Address hostaddress = new Address();
    hostaddress.ServiceProvider = Address.ServiceProviderTcpIp;
    hostaddress.AddComponent(Address.KeyPort, SharedCode.DataPort);
    // Set our name
    PlayerInformation info = new PlayerInformationf);
    info.Name = textBoxl.Text;
    connection.SetClientInformation(info, SyncFlags.ClientInformation);
    // Set up an application description
    ApplicationDescription desc = new ApplicationDescription();
    desc.GuidApplication = SharedCode.ApplicationGuid;
    try
     {
       // Search for a server
       connection.FindHosts(desc, hostaddress, deviceAddress, null,
          0, 0, 0, FindHostsFlags.None);
          AddText("Looking for sessions.");
     }
```

```
    catch
    {
       AddText("Enumeration of sessions has failed.");
    }
}
```

Здесь имеется много общего с тем, что мы уже делали в предыдущей главе. После создания клиента приложение производит поиск хостов, затем пытается установить соединение с ними, затем проводится проверка наличия службы поддержки доступа. Эти вопросы мы уже рассматривали при написании программы для P2P-сети.

Далее создаются адреса TCP/IP для локальной машины и отдельный адрес для сервера. Как мы уже знаем, любой главный компьютер, к которому мы будем подключаться, будет задействовать порт, определенный в SharedCode.DataPort. Добавим этот компонент, а также имя хоста (если оно нам известно) к адресу сервера.

Каждый "участник" сеанса в DirectPlay может иметь некоторую специфическую информацию, которая может содержать имя "игрока" или описание приложения. Например, если мы хотим изменить имя клиентского приложения, которым на данный момент является слово "textboxl", мы можем сделать это с помощью следующего выражения:

```
textBoxl.Text  =  System.Environment.UserName;
```

Эта строка устанавливает в качестве имени клиентской программы внутреннее имя нашего компьютера. Способ изменения установленного имени пользователя будет рассмотрен ниже.

ПОИСК В ЛОКАЛЬНОЙ ПОДСЕТИ

- Обратите внимание, что в этом примере мы не использовали флажок FindHostsFlags.OkToQueryForAddressing. Мы полагали, что сервер находится в той же самой подсети (значение по умолчанию для DirectPlay). В противном случае мы можем использовать этот флажок или непосредственно добавить имя сервера в качестве компонента его адреса.

Затем мы вызываем процедуру описания приложения. Обратите внимание, что единственным параметром, который мы устанавливаем в этом случае, является идентификатор GUID. Этот идентификатор должен быть одним и тем же и для сервера, и для клиента. И, наконец, выполняется вызов метода FindHosts, сопровождающийся надписью «поиск соединения».

Теперь нам понадобятся две процедуры установки и завершения соединения, которые мы объявили вначале. Соответствующий код приведен в листинге 19.3.

Листинг 19.3. Обработчик для установки и завершения соединения.

```
private void OnFindHost(object sender, FindHostResponseEventArgs e)
{
   lock(this)
   {
      // Do nothing if we're connected already
      if (connected)
         return;
      connected = true;
      string foundSession = string.Format
         ("Found session ({0}), trying to connect.",
         e.Message. ApplicationDescription. SessionName) ;
      this.BeginInvoke(new AddTextCallback(AddText),
         new object[] { foundSession });
      // Connect to the first one
      ((Client)sender).Connect(e.Message.ApplicationDescription,
         e.Message.AddressSender,
         e.Message.AddressDevice, null, ConnectFlags.OkToQueryForAddressing);
   }
}
private void OnConnectComplete(object sender, ConnectCompleteEventArgs e)
{
   // Check to see if we connected properly
   if (e.Message.ResultCode == ResultCode.Success)
   {
      this.BeginInvoke(new AddTextCallback(AddText),
         new object[] ( "Connect Success." lb-
      connected = true;
      this.BeginInvoke(new EnableCallback(EnableSendDataButton),
         new object[] { true ) );
   }
   else
   {
      this.BeginInvoke(new AddTextCallback(AddText), new object[] {
         string.Format("Connect Failure: {0}", e.Message.ResultCode) });
      connected = false;
      this.BeginInvoke(new EnableCallback(EnableSendDataButton),
         new object[] ( false } );
   }
}
```

Методы достаточно знакомы нам по сценариям P2P-соединения. Вначале мы пытаемся найти в сети доступные нам серверы, после чего пытаемся к ним подключиться. При завершении соединения соответствующий обработчик отслеживает это событие. Вызов операции dispose в данном случае не используется, поскольку это бышо сделано при написании серверной части (вернее, общего для обеих частей фрагмента кода). Теперь необходимо переписать наш основной метод следующим образом:

```
static void Main()
{
   using (Forml frm = new Forml())
   {
      frm.Show();
      frm.InitializeClient();
      Application.Run(frm);
   }
}
```

Таким образом, наш «клиент» готов к работе. Теперь можно запустить оба приложения. Мы должны увидеть, что сервер сообщит нам о начале работы, а клиент после обнаружения сервера присоединится к нему. Попробуем теперь слегка разнообразить наши приложения.

Отслеживание моментов присоединения и выхода из сети

Теперь попробуем отследить моменты, когда клиенты соединяются с сервером, и когда они покидают сеть. Собственно, нам необходимо создать две процедуры обработки этих событий. Вначале добавьте в методе инициализации для сервера следующие обработчики:

```
connection.PlayerCreated += new
   PlayerCreatedEventHandler(OnPlayerCreated);
connection.PlayerDestroyed += new
   PlayerDestroyedEventHandler(OnPlayerDestroyed);
```

Событие регистрируется всякий раз, когда игрок присоединяется или покидает сеть. В листинге 19.4 приведен код указанных процедур, добавьте его в приложение для сервера.

Листинг 19.4. Обработчик клиента.

```
private void OnPlayerCreatedfobject sender, PlayerCreatedEventArgs e)
 {
```

```
    try
    {
        string playerName = ((Server)sender).GetClientlnforaation
            (e.Message.PlayerlD).Name;
        string newtext = string.Format
            ("Accepted new connection from (0), UserlD: 0x{l}",
            playerName, e.Message.PlayerlD.ToString("x")) ;
        this.Beginlnvoke(new AddTextCallback(AddText),
            new object[] ( newtext ));
    }
    catch { /* Ignore this, probably the server */ }
}
private void OnPlayerDestroyed(object sender, PlayerDestroyedEventArgs e)
{
    string newtext = string.Format
        ("DirectPlayer UserlD: 0x{0} has left the session.",
        e.Message.PlayerlD.ToString("x"));
    this.Beginlnvoke(new AddTextCallback(AddText),
        new object[] ( newtext ));
}
```

Процедура обработки выхода из сети достаточно проста, и в результате ее выполнения выводится сообщение о том, что пользователь покинул сеть.

Что же касается алгоритма отслеживания момента входа игрока в сеть, то он еще не был рассмотрен нами ни разу. Все это происходит следующим образом: в клиентской части приложения устанавливается имя клиента и вызывается метод SetClientlnformation; в серверной части мы ищем это имя с помощью процедуры GetClientlnfomiation. При первом запуске сервера по умолчанию создается «скрытый» игрок, и, пока не появится соответствующий ему клиент, процедура SetClientlnformation игнорируется.

КОНТРОЛЬ ДОСТУПА В СЕТЬ

Вполне возможно, что вам может понадобиться запретить доступ в сеть для некоторых пользователей (например, даже для тех, кто уже знает имя сервера, идентификатор GUID и порт).

Помимо обработчика операции входа в сеть имеется функция IndicateConnect, аргументом которой (IndicateConnectEventArgs) является параметр RejectMessage (по умолчанию со значением «false»). Опираясь на значение параметра этого обработчика, мы можем либо позволить, либо не позволить выполнение этого соединения. В первом случае нам не нужно ничего предпринимать дополнительно, сеанс выполнится автоматически. Во втором случае мы просто устанавливаем параметр RejectMessage в значение «true».

Что если мы вообще хотим сделать вызов сервера недоступным? Как нам известно, при нахождении сервера или хоста пользователь принимает информацию FindHostsResponse, но перед этим сервер в свою очередь получит обращение FindHostsQuery. Таким образом, мы можем использовать это событие, отследив его и применив уже описанную стратегию. Используя комбинации соответствующих процедур, мы можем осуществить полный контроль всей сети.

Следует отметить, что хост-компьютеры для пиринговых сетей имеют аналогичные возможности.

Передача пакетов данных по сети

В главе 19, описывая P2P-сети, мы уже создали некоторую процедуру для передачи простого пакета данных, если быть точнее, одного байта данных. При работе сетей «клиент-сервер» трафик обмена данными может быть весьма значительным.

Для моделирования реального приложения мы можем добавить к клиентскому приложению несколько опций, которые позволят облегчить отправку пакетов данных на сервер. Для проверки мы можем выполнить несколько стандартных действий.

Определить число игроков в сети.

- Послать сообщения другим игрокам, находящимся в сети.
- Послать сообщение на сервер.
- Стать недоступным для других участников сети.
- Изменить имя клиента.

Очевидно, что и клиентское, и серверное приложения должны «знать» об этих действиях, для этого необходимо добавить перечень указанных действий к файлу кода ShareCode:

```
   public  enum  NetworkMessages
{
      CheckPlayers,
      Wave,
      SendData,
      RunAway,
      ChangeName,
   }
```

Теперь у нас имеется отдельное значение для каждого действия. Для того чтобы клиент мог выполнить указанные действия, создайте пять соответствующих кнопок в верхней части нашей формы. Для упрощения программы используем одну процедуру для всех пяти кнопок. Этот метод приведен в листинге 19.5.

Листинг 19.5. Обработка кнопок для отправки пакетов данных.

```
private void button_Click(object sender, System.EventArgs e)
 {
    // We want to send some data to the server
    NetworkPacket packet = new NetworkPacket();
    // Write the correct message depending on the button pressed
    switch(((Button)sender).Name)
    {
       case "button1":
       packet.Write(NetworkMessages.CheckPlayers);
       break;
    case "button2":
       packet.Write(NetworkMessages.Wave);
       break;
    case "button3":
       packet.Write(NetworkMessages.SendData);
       break;
    case "button4":
       packet.Write(NetworkMessages.RunAway);
       break;
    case "button5":
       if (textBox1.Text.Length > 0)
       {
          packet.Write(NetworkMessages.ChangeName);
          packet.Write(textBox1.Text);
          PlayerInformation info = new PlayerInformation();
          info.Name = textBox1.Text;
          connection.SetClientInformation(info,
          SyncFlags.ClientInformation);
       }
       else
       {
          // Don't try to do anything if there is no name
          return;
       }
       break;
    }
    connection.Send(packet, 0, SendFlags.Guaranteed);
 }
```

В зависимости от предпринятых действий, в сетевой пакет записывается соответствующее сообщение. Если мы нажмем кнопку «ChangeName» и введем некоторый текст в диалоговое окно, мы не только запишем это сообщение в пакет данных, но и изменим соответствующим образом информацию о клиентском приложении.

Наконец, данные отправляются на сервер. Обратите внимание, что здесь нет никакого пользовательского идентификатора для клиента, поскольку в соединении клиент-сервер клиент может посылать данные только на сервер.

Теперь, подготовив клиентскую часть к отправке данных, необходимо подготовить сервер к приему и обработке этих данных. Естественно, в первую очередь мы должны зафиксировать событие поступления данных на сервер. Добавьте следующую строку к методу инициализации сервера:

```
connection.Receive += new
   ReceiveEventHandler(OnDataReceive);
```

Закончить процедуру обработки этого события весьма не сложно. Мы должны определить тип данных и среагировать на них соответствующим образом, см. листинг 19.6.

Листинг 19.6. Обработчик для принимаемых на сервере данных.

```
private void OnDataReceive(object sender, ReceiveEventArgs e)
{
   NetworkMessages msg = (NetworkMessages)e.Message.ReceiveData.Read
    (typeof(NetworkMessages));
   NetworkPacket returnedPacket = new NetworkPacket0;
   string newtext = string.Empty;
   switch (msg)
   {
      case NetworkMessages.ChangeName:
         string newname = e.Message.ReceiveData.ReadString0;
         newtext = string.Format
             ("DPlay Userld 0x{0| changed name to {1}",
            e.Message.SenderlD.ToStringPx"), newname);
         // The user wants inform everyone they ran away
         returnedPacket.Write(NetworkMessages.ChangeName);
         returnedPacket.Write(e.Message.SenderlD);
         returnedPacket.Write(newname);
         // Now send it everyone
         connection.SendTo((int)PlayerID.AHPlayers, returnedPacket, 0,
            SendFlags.Guaranteed ! SendFlags.NoLoopback);
         break;
         case NetworkMessages.CheckPlayers:
            newtext = string.Format
                ("Received CheckPlayers from DPlay Userld: Ox{0}",
               e.Message.SenderlD.ToString("x"));
         // The user wants to know how many players are in the session
```

```
            returnedPacket.Write(NetworkMessages.CheckPlayers);
            // subtract one user for the server user
            returnedPacket.Write(connection.Players.Count - 1);
            // Now send it only to that player
            connection.Sendlofe.Message.SenderlD, returnedPacket, 0,
               SendFlags.Guaranteed);
            break;
         case NetworkMessages.RunAway:
            newtext = string.Format
                ("Received RunAway from DPlay Userld: 0x{0}",
               e.Message.SenderlD.ToString("x"));
            // The user wants inform everyone they ran away
            returnedPacket.Write(NetworkMessages.RunAway);
            returnedPacket.Write(e.Message.SenderID);
            // Now send it everyone
            connection.SendTo((int)PlayerID.AllPlayers, returnedPacket, 0,
               SendFlags.Guaranteed | SendFlags.NoLoopback);
            break;
         case NetworkMessages.SendData:
            newtext = string.Format
                ("Received SendData from DPlay Userld: 0x{0}",
               e.Message.SenderlD.ToString("x"));
            // No need to reply, 'fake' server data
            break;
         case NetworkMessages.Wave:
            newtext = string.Format
                ("Received Wave from DPlay Userld: 0x{0}",
               e.Message.SenderlD.ToString("x"));
            // The user wants inform everyone they waved
            returnedPacket.Write(NetworkMessages.Wave);
            returnedPacket.Write(e.Message.SenderlD);

               // Now send it everyone
               connection.SendTo( (int)PlayerlD.AHPlayers, returnedPacket, 0,
                  SendFlags.Guaranteed \ SendFlags.NoLoopback);
               break;
         }
         // We received some data, update our UI
         this.Beginlnvoke(new AddTextCallback(AddText),
            new object[] ( newtext });
   }
```

Этот метод не так сложен, как может показаться. Одним из его параметров является передаваемый сетевой пакет. Аналогично вызову Write (когда мы размещали данные в пакет) для извлечения этих данных вызывается метод Read.

ЧТЕНИЕ ДАННЫХ В УСТАНОВЛЕННОМ ПОРЯДКЕ

Обратите внимание, что данные должны читаться в том же самом порядке, в котором они были записаны. В противном случае при работе приложения могут возникнуть ошибки.

Метод Read должен иметь в качестве входного параметра тип полученных нами данных, который мы должны предварительно определить. Затем мы устанавливаем необходимость отправки ответа либо определенному клиенту, либо всем участникам сети.

Для отправки данных мы используем самое простое действие — SendData. Данная процедура не требует обратной связи (за исключением сообщения собственно на самом сервере).

Действия Wave и RunAway идентичны друг другу. При получении любого из этих сообщений вначале, как обычно, обновляется пользовательский интерфейс на сервере, далее формируется новый пакет данных, куда записывается полученная информация и идентификатор пославшего его пользователя. Этот пакет рассылается всем участникам сеанса связи. Обратите внимание, что при отправке данных с сервера мы используем флажок NoLoopBack, который позволяет избежать получения ответных сообщений.

Действие CheckPlayers отличается от остальных. Клиентское приложение запрашивает на сервере необходимую только ему информацию. Сервер снова записывает идентификатор сообщения в новый сетевой пакет, затем указывает число игроков, находящихся в сети (исключая виртуальных игроков на сервере), и, наконец, отправляет новый пакет непосредственно отправителю запроса, так что никакие другие игроки в сети не могут увидеть это сообщение (если только они не запрашивают ту же самую информацию). В завершение на сервере обновляется пользовательский интерфейс.

Сценарий последнего действия похож на сценарии Wave и RunAway. С помощью метода ReadString мы извлекаем информацию о новом имени клиента непосредственно из сетевого пакета. Единственное главное отличие от описанных выше сценариев Wave и Run away заключается в том, что после включения в новый пакет данных идентификаторов сообщения и отправителя, мы также записываем туда новое имя клиента, инициировавшего указанное действие.

Формирование отклика клиента

Теперь мы можем написать процедуру, позволяющую отследить принятие данных клиентом. Добавьте в приложение Client фрагмент кода, похожий на тот, который мы использовали для тех же целей на сервере, см. листинг 19.7.

Листинг 19.7. Обработчик получения данных клиентом.

```
private void OnDataReceive(object sender, ReceiveEventArgs e)
{
   NetworkMessages msg = (NetworkMessages)e.Message.ReceiveData.Read
       (typeof(NetworkMessages));
   string newtext = string.Empty;
   int playerlD = 0;
   switch (msg)
   {
      case NetworkMessages.ChangeName:
         playerlD = (int)e.Message.ReceiveData.Readftypeof(int));
         string newname = e.Message.ReceiveData.ReadString();
         newtext = string.Format
             ("DPlay Userld 0x(0( changed name to {1}",
            playerlD.ToString(Y), newname);
         break;
      case NetworkMessages.CheckPlayers:
         int count = (int)e.Message.ReceiveData.Readftypeof(int));
         newtext = string.Format
             ("Server reports {0} users on the server currently.",
            count);
         break;
      case NetworkMessages.RunAway:
         playerlD = (int)e.Message.ReceiveData.Readftypeof (int));
         newtext = string.Format
             ("Server reports DPlay Userld 0x{0} has ran away.",
            playerlD.ToString(Y));
         break;
      case NetworkMessages.Wave:
         playerlD = (int)e.Message.ReceiveData.Readftypeof(int));
         newtext = string.Format
             ("Server reports DPlay Userld 0x{0} has waved.",
            playerlD.ToString(Y));
         break;
   )
   //We received some data, update our UI
   this.Beginlnvoke(new AddTextCallback(AddText),
      new object[] { newtext });
}
```

Как вы можете видеть, по сути, это упрощенная версия соответствующей серверной процедуры. При получении сообщения обновляется интерфейс, уведомляющий пользователя о результате выполненных действий.

Обработка отключения сервера

Последнее, что мы должны сделать для созданного соединения, — правильно отследить момент потери или отсоединения сервера. Нечто подобное мы уже проделывали при работе с P2P-сетями. Вначале мы добавляем определение этого события (SessionTerminated):

```
connection.SessionTerminated += new
    SessionTerminatedEventHandler(OnSessionTerminate);
```

И записываем код для этого обработчика:

```
private void OnSessionTerminate(object sender, SessionTerminatedEventArgs
e)
{
    this.Beginlnvoke(new DisconnectCallback(OnDisconnect), null);
}
private void OnDisconnect()
{
    EnableSendDataButtons(false);
    AddText("Session terminated.");
    connected = false;
    // Dispose of our connection, and set it to null
    connection.Dispose();
    connection = null;
}
```

Данная процедура проверяет факт отсоединения и освобождает объект. Также выводится сообщение о том, что соединение разорвано.

Краткие выводы

В этой главе мы рассмотрели следующие вопросы.

- **Создание выделенных серверов.**
- **Соединение с серверами с помощью клиентского интерфейса.**
- **Отслеживание подключения и отключения игрока.**
- **Передача игровых данных.**

В следующей главе мы изучим более совершенные методы, используемые в сетевых приложениях, включая определение пропускной способности, статистику, запуск приложений и даже голосовую связь.

Глава 20. Особенности более совершенного использования сетей

Эта глава охватывает дополнительные возможности DirectPlay, которые позволят нам расширить диапазон использования сетей, и включает следующие разделы.

- Модели событий.
  Пропускная способность, трафик.
- Очередность отправки данных.
  Приложения лобби «lobby-launching».
  Использование голосовой связи.

Модели событий и обработчики

К настоящему моменту мы изучили работу P2P-сетей и основы архитектуры клиент-сервер. Теперь мы можем попытаться расширить диапазон наших знаний относительно использования сетей.

До сих пор мы рассматривали лишь некоторые из возможных в течение сеанса связи событий. Ниже приведен список практически всех событий, доступных для обработки с помощью DirectPlay, с указанием классов, для которых применимы данные процедуры.

ApplicationDescriptionChanged — применяется для классов Peer, Server и Client. Отслеживается событие, когда изменяется описание объекта (например, меняется имя).

AsyncOperationComplete — применяется для классов Peer, Server и Client. Фиксирует завершение асинхронной операции. Некоторые функции DirectPlay (например, Connect или Receive) возвращают параметр, который сообщает приложению о завершении выполнения. С помощью этого параметра соответствующие методы можно отменить, используя процедуру CancelAsyncOperation. Необходимо помнить, что некоторые операции могут заканчиваться автоматически, как, например, метод FindHosts при установке соединения.

- ConnectComplete — применяется для классов Peer и Client. Событие возникает после неудачной попытки соединения. Значение параметра ResultCode указывает на причину возникшей ситуации. Например, если хост отклонил ваш запрос о подключении, результатом будет значение HostRejectedConnection.

  FindHostQuer — применяется для классов Peer и Server. Событие фиксируется, когда новое клиентское приложение вызывает метод FindHosts и начинает выполнять поиск хоста. Если вы не измените значение параметра RejectMessage (по умолчанию «false») перед тем, как ответить на запрос, метод FindHosts успешно обна-

ружит ваш хост. Установка аргумента RejectMessage в значение «true» сделает ваш сеанс недоступным для клиента.

- FindHostResponse — применяется для классов Peer и Client. Отслеживается событие обнаружения хоста. Возвращаемый параметр будет содержать информацию (адрес и описание хоста), достаточную для того, чтобы соединиться с найденным сервером. Если не происходит немедленного соединения с сервером, поиск будет продолжаться, и вполне возможно, что один и тот же хост будет найден несколько раз.

  GroupCreated — применяется для классов Peer и Server. Возникает в результате успешного выполнения процедуры создания групп игроков CreateGroup.
- GroupDestroyed — применяется для классов Peer и Server. Отслеживается событие расформирования группы (например, в результате использования метода DestroyGroup или при разрыве соединения). Если при создании группы использовался флажок AutoDestruct, группа будет автоматически расформирована, если один из участников завершил сеанс. Причину расформирования группы можно найти в параметре Reason.
- GroupInformation — применяется для классов Peer и Server. Когда изменяется информация о группе (например, при помощи метода SetGroupInformation), это событие регистрируется всеми ее участниками. В качестве параметра возвращается только идентификатор группы, поэтому не забудьте запросить остальную информацию.
- HostMigrated — обработчик применим только к классу Peer. Если в течении P2P-сеанса главный компьютер вышел из соединения, статус хоста передается следующему компьютеру. Событие происходит, только если при создании хоста был включен соответствующий флажок.
- IndicateConnect — применяется для классов Peer и Server. Событие, аналогичное событию FindHostQuery (соединение также может быть запрещено).
- IndicateConnectAborted — применяется для классов Peer и Server. Как правило, за событием IndicateConnect должно следовать событие PlayerCreated. Однако, если по каким-либо причинам соединение было прервано до этого момента (например, случайный разрыв соединения), произойдет указанное событие.
- PeerInformation — обработчик применим к классу Peer. Отслеживает изменение данных объекта peer. При необходимости следует восстановить остальную информацию.
- PlayerAddedToGroup — применяется для классов Peer и Server. Происходит, когда новый игрок (или группа игроков) добавляется к существующей группе. Использует идентификаторы группы и игрока.

- PlayerCreated — применяется для классов Peer и Server. Данное событие следует за событием IndicateConnect. В момент выполнения обработчика в качестве аргумента можно устанавливать контекстную переменную для игрока (любая специфическая информация об игроке). Каждый раз, получая данные от игрока, можно получать и контекстную переменную этого игрока.
- PlayerDestroyed — применяется для классов Peer и Server. Событие фиксируется, когда игрок покидает сеанс связи. Когда же завершается сам сеанс связи, мы получаем подобное сообщение от каждого клиента.

  PlayerRemovedFromGroup — применяется для классов Peer и Server. Происходит при удалении игрока из группы (например, с помощью вызова RemovePlayerFromGroup) или выходе игрока из сеанса.

  Receive — применяется для классов Peer, Server и Client. Информирует о получении новых данных (сетевого пакета, идентификатора игрока, контекстной переменной игрока, отправляющего информацию и т. д.). Обратите внимание на использование флажка NoLoopback, который мы уже описывали раньше.
- SendComplete — применяется для классов Peer, Server и Client. Возникает в результате завершения асинхронной операции передачи данных. Если используется флажок CompleteOnProcess, событие будет зарегистрировано только после получения сообщения объектом, которому вы его направляете. Если используется флажок NoComplete, событие не будет обработано.

  SessionTerminated — применяется для классов Peer и Client. Событие отслеживается каждый раз при завершении сеанса. Получить информацию о причине завершения соединения можно с помощью параметра ResultCode. Может содержать определяемую пользователем информацию (например, для организации другого подключения)

  ClientInformation — применяется для классов Server и Client. Отслеживается момент изменения информации о клиенте. При необходимости информацию следует восстановить после регистрации события.
- ServerInformation — применяется для классов Server и Client. Отслеживает изменение информации о сервере. При необходимости требуется восстановление информации после регистрации события.

Определение пропускной способности и статистики сети

На сегодняшний день вопросы пропускной способности соединений являются весьма актуальными. Подключение к Internet с помощью моде-

ма имеет достаточно низкую пропускную способность. Сеть на базе Ethernet более предпочтительна и является наиболее распространенной на данный момент, но и здесь также имеются ограничения. Задача разработчика сетевых приложений состоит в том, чтобы самым рациональным образом распределять трафик, добиваясь максимальной пропускной способности соединения.

При написании серверной части для сетевой игры необходимо предусмотреть максимальную нагрузку на канал сервера, поскольку подразумевается поддержка и обмен данными с тысячами пользователей одновременно. В этом случае работа над пропускной способностью сервера отличается от той же работы для клиента.

DirectPlay API достаточно универсален и позволяет оптимальным образом распределять пересылаемые клиентам данные. Для получения информации о состоянии имеющихся подключений необходимо, указав идентификатор запрашиваемого игрока, вызвать метод GetConnectionInformation. Клиент возвратит информацию о соединении с сервером. Ниже приведен пример такой информации:

```
Connection information:
PacketsDropped: 0
BytesDropped: 0
PacketsRetried: 0
BytesRetried: 0
PacketsSentNonGuaranteed: 1
BytesSentNonGuaranteed: 0
PacketsSentGuaranteed: 99
BytesSentGuaranteed: 1344
PeakThroughputBps: 116
ThroughputBps: 80
RoundTripLatencyMs: 4
MessagesReceived: 79
PacketsReceivedNonGuaranteed: 0
BytesReceivedNonGuaranteed: 0
PacketsReceivedGuaranteed: 79
BytesReceivedGuaranteed: 1836
MessagesTimedOutLowPriority: 0
MessagesTransmittedLowPriority: 0
MessagesTimedOutNormalPriority: 0
MessagesTransmittedNormalPriority: 99
MessagesTimedOutHighPriority: 0
MessagesTransmittedHighPriority: 1
```

В этом блоке содержится практически вся информация о соединении: число пакетов, принятых, отправленных, имеющих различные статусы и флажки и т. д.

13 Зак. 604

Здесь же можно найти информацию о максимальной (PeakThroughputBps) и средней (ThroughputBps) пропускной способности канала, измеряемой в байтах за секунду. Данная информация позволяет проверить, насколько оптимально загружается канал.

Под средним временем ожидания (RoundTripLatencyMs) можно понимать время, затраченное на отправку пакета и на пересылку его обратно. Данный параметр коррелирует с пропускной способностью, но при этом позволяет отследить временные затраты для данного конкретного канала.

В завершение мы получаем информацию о количестве сообщений, полученных в различные моменты времени и имеющих различные приоритетные уровни (нормальный, низкий и высокий).

Необходимо отметить, что DirectPlay не будет посылать сообщения удаленному компьютеру быстрее, чем тот может их обработать. Если удаленный компьютер не отвечает долгое время, отправитель создаст очередность отправляемых данных. Перед отправкой данные могут быть сгруппированы в пакеты, которые, в свою очередь, могут быть объединены. Вы можете определить количество данных в очереди, вызывая метод GetSendQueueInformation (выполняется аналогично методу GetConnectionInformation для сервера или Peer-объекта). При этом запрос возвращает два целых числа (для каждого из объектов), первое — число сообщений, находящихся в очереди, и второе — суммарное число байтов для этих сообщений.

Также возможно управлять приоритетом отсылаемых данных, используя соответствующие флажки GetSendQueueInformationFlags. Это позволяет отследить момент переполнения очереди и ограничить количество пересылаемых данных.

Наибольшее внимание при настройке соединений следует уделить устранению «ненужных» данных, которые передаются по каналу (это касается пересылки без необходимости строк, булевых переменных и пр.). Вместо того чтобы отправить четыре логические переменные, попробуйте отправить один байт с побитовой маскировкой для различных булевых констант. Например:

```
private bool IsRunning = true;
private bool IsMale = false;
private bool IsWarrior = true;
private bool IsGhost = false;
```

Если отправлять данные в таком виде, передаваемый пакет будет иметь размер, равный 16-ти байтам, что является весьма расточительным в плане трафика. Наша задача снизить это значение, по крайней мере, до одного байта:

```
private const byte IsRunning = Oxl;
private const byte IsMale = 0x2;
private const byte IsWarrior = 0x4;
private const byte IsGhost = 0x8;
private byte SendData = IsRunning | IsWarrior;
```

Теперь однобайтовая переменная SendData содержит информацию, соответствующую четырем булевым переменным. Таким способом мы можем «замаскировать» до 8-ми логических переменных (экономя при этом 31 байт). Если необходимо переслать более чем 8 переменных, можно использовать короткий формат, который может поддерживать до 16-ти булевых переменных (экономя 62 байта) или длинный формат, который может поддерживать до 64-х булевых переменных (экономя 252 байта). Для уменьшения трафика при возможности необходимо использовать наименьший формат.

Запуск приложений, использующих концепцию «Lobby»

Для того, кто когда-либо запускал игры типа MSN Games (сайт http://zone.msn.com), концепция «лобби» достаточно понятна. По существу, лобби — это подход, когда группа игроков собирается перед запуском игры, и игра запускается одновременно для всех игроков, при этом все игроки автоматически соединяются друг с другом.

Вспоминая сборки для DirectPlay, необходимо заметить, что это — первая используемая нами сборка, имеющая подпространство имен. Внутри этой сборки имеются два дополнительных пространства имен: Lobby и Voice. Свойства второго списка мы рассмотрим позже, а пока сконцентрируемся на пространстве имен Lobby.

Пространство Lobby включает два основных класса, которые будут управлять взаимодействиями в лобби-приложениях: Application и Client. Класс Client используется для запуска и поддержки лобби-приложений на удаленных машинах, а класс Application непосредственно управляет лобби-приложением. Каждый из упомянутых классов имеет модели событий, подобно тем моделям, которые используются в классах Peer, Server и Client.

Вначале нам необходимо определить программы, которые могут запускаться в режиме Lobby, а уже потом попробовать запустить их. Как обычно, создаем новое окно приложения, устанавливаем параметр Dock в значение «Fill», проверяем добавление ссылок на DirectPlay и директиву using для пространства имен Lobby:

```
using Microsoft.DirectX.DirectPlay.Lobby;
```

Отдельно объявляем переменную класса Client, поскольку именно этот класс отвечает за объекты и запуск лобби-приложения:

```
private Client connection = null;
```

Так же мы должны освободить объект после выхода из соединения, добавив две строки к соответствующей перегрузке Dispose:

```
if (connection != null)
   connection.Dispose ();
```

Следует отметить, что имя класса Client встречается и в пространстве имен DirectPlay, и в пространстве имен Lobby. Таким образом, мы должны однозначно определить оба этих пространства и включить их в директиву using. Это же касается и другого класса (Application), который уже включен в пространство имен System.Windows.Forms.

Теперь давайте заполнять окно списка «list box» для наших лобби-приложений. Добавьте следующий код к вашему конструктору форм:

```
// Fill the list box
connection = new Client();
foreach(ApplicationInformation ai in connection.GetLocalPrograms())
{
   listBoxl.Items.Add(ai.ApplicationName);
}
```

Как видите, это относительно простая операция. После создания лобби-объекта Client происходит перебор всех имеющихся локальных программ, и название каждой из них записывается в наше окно. Запуск этих приложений не сложен; добавим процедуру обработки двойного нажатия кнопки мыши в окне списка. Для этого используем код, приведенный в листинге 20.1.

Листинг 20.1. Запуск приложения.

```
private void HstBoxl_DoubleClick(object sender, System.EventArgs e)
}
   if (listBoxl.Selectedltem == null)
      return;
   foreachfApplicationlnformation ai in connection.GetLocalPrograms())
   {
      if (ai.ApplicationName == (string)listBoxl.Selectedltem)
      {
         Connectlnformation ci = new Connectlnformation();
```

```
            ci.GuidApplication = ai.GuidApplication;
            ci.Flags = ConnectFlags.LaunchNew;
            connection.ConnectApplication(ci,
                System.Threading.Timeout.Infinite, null);
            break;
        }
    }
}
```

Этот фрагмент кода — не самый эффективный, поскольку нам придется проходить список дважды, но весьма показательный. Если имеется выбранный нами пункт, мы находим его, после чего инициализируется структура ConnectInformation с идентификатором GUID и управляющими флажками.

ПРИСОЕДИНЕНИЕ К С СУЩЕСТВУЮЩЕЙ СЕССИИ ИЛИ СЕАНСУ

Мы также можем присоединиться к уже выполняемому приложению, имеющему статус лобби. Для этого необходимо установить флажок ConnectFlags.LaunchNotFound в структуре ConnectionSettings (если действующее приложение не будет найдено, запустится новый экземпляр).

Следует отметить, что при компиляции появится сообщение о неопределенности класса Application, поскольку класс Application встречается и в System.Windows.Forms, и в Microsoft.DirectX.DirectPlay.Lobby. По этой причине перепишем основную процедуру следующим образом:

```
static void Main()
{
    using (Form1 frm = new Form1())
    {
        frm.Show();
        System.Windows.Forms.Application.Run(frm);
    }
}
```

После успешного запуска приложения метод ConnectApplication возвращает обработчик соединения, с помощью которого мы можем отправить данные в приложение, используя метод Send, или вызывать метод ReleaseApplication, который завершит работу приложения (но это приведет не к закрытию приложения, а к отсоединению лобби-клиента от данного приложения).

Создание лобби-приложения

Класс Application в пространстве имен Lobby определяет структуру информации, доступную после запуска лобби-сеанса. Регистрация лобби-приложения довольно проста. Необходимо создать и заполнить структуру ProgramDescription, обязательно указав идентификатор GUID для этого приложения, имя, путь и параметры файла. Затем мы можем вызвать либо метод RegisterProgram, либо метод UnregisterProgram в зависимости от выполняемой операции.

До сих пор мы подробно не рассматривали конструктор объекта Application для лобби-соединения. Обсудим в качестве примера вариант, содержащий наибольшее число параметров:

```
public Application ( System.Int32 connectionHandle ,
    Microsoft.DirectX.DirectPlay.Lobby.InitializeFlags flags ,
    Microsoft.DirectX.DirectPlay.Lobby.ConnectEventHandler connectEventHandler
)
```

Параметр обработчика подключения является выходным параметром. Если ваше приложение было запущено в качестве лобби-клиента, этот обработчик будет возвращен процедурой ConnectApplicaton, в противном случае он будет не определен. Использование флажков InitializeFlags позволяет отключить проверку этого параметра. Последний параметр *ConnectEventHandler* — позволяет отслеживать событие подключения еще до создания объекта.

После запуска лобби-клиента необходимо вызвать метод RegisterLobby для подключения к соответствующему сеансу, используя в качестве параметра структуру ConnectionSettings, полученную с помощью обработчика подключения.

Добавление голосового чата

Добавление комментариев к игре в режиме текстовых сообщений является достаточно распространенным приемом. Тем не менее, даже при очень большой скорости набивания текста это не совсем удобно. Таким образом, возникает необходимость в голосовом способе передачи сообщений, позволяющем не отрывать игрока от процесса игры.

Добавление голосового чата в приложение сервер-клиент не представляет огромного труда. Возьмем пример из главы 18, где мы описывали P2P-соединение, и попробуем добавить голосовое общение.

Подготовив предварительно наш новый проект, мы должны добавить ссылку на DirectSound, а также на соответствующие переменные и директивы пространства имен Voice (учитывая, что данное пространство используется и в классе Server, и в классе Client):

```
using Voice = Microsoft.DirectX.DirectPlay.Voice;
using  Microsoft.DirectX.DirectSound;
```

Объявляем переменные Voice для сервера и клиента. Для P2P-сети главный компьютер (хост) играет роль сервера, но поскольку любой из объектов P2P-сети может принять полномочия хоста, объект peer должен содержать обе части Voice — и для сервера, и для клиента:

```
private Voice.Client voiceClient = null;
private Voice.Server voiceServer = null;
```

Как и раньше, вначале мы должны создать серверную часть. Необходимо найти и переписать секцию кода, где мы создавали хост для P2P-сети. Затем в методе инициализации InitializeDirectPlay следует добавить следующий код после вызова EnableSendDataButton:

```
// Create our voice server first
voiceServer = new Voice.Server(connection);
// Create a session description
Voice.SessionDescription session = new Voice.SessionDescriptionf);
session.SessionType = Voice.SessionType.Peer;
session.BufferQuality = Voice.BufferQuality.Default;
session.GuidCompressionType = Voice.CompressionGuid.Default;
session.BufferAggressiveness = Voice.BufferAggressiveness.Default;
// Finally start the session
voiceServer.StartSession(session);
```

Итак, теперь мы должны связать создаваемый объект Voice с объектом DirectPlay, который будет выполнять функцию передачи голоса. Это позволит передавать голос наряду с другими данными сети. Поскольку мы используем P2P-сеть, сообщения будут передаваться другим игрокам напрямую, однако существуют и другие сценарии голосового общения.

- Смешанный — в этом режиме все голосовые сообщения отсылаются серверу. Затем сервер объединяет их и переправляет игрокам. Такой подход перегружает трафик, но является очень удобным.
- Перенаправляющий — и в этом режиме все голосовые сообщения проходят через хост. Такой вариант снижает трафик для каждого клиента, но значительно увеличивает его для главного компьютера. Если пропускная способность главного компьютера мала, этот сценарий будет бесполезным.

  Эхо возвращает звуковое сообщение назад.

Еще одной опцией, которую мы можем установить в описании соединения, является кодек. В нашем приложении кодек установлен по умолчанию, а ниже приводится фрагмент кода, позволяющий распечатать в окне вывода кодеки, которые можно использовать дополнительно:

```
foreach(Voice.CompressionInformation ci in voiceServer.CompressionTypes)
{
    Console.WriteLine(ci.Description);
}
```

Данная информация позволит выбрать наиболее подходящий кодек. Вы можете также использовать любой из кодов сжатия, перечисленных в классе CompressionGuid.

Для оставшихся параметров мы устанавливаем значения по умолчанию.

И как обычно, мы добавляем процедуру выхода из соединения и освобождения объекта:

```
if (voiceServer != null)
{
    voiceServer.StopSession();
    voiceServer.Dispose();
    voiceServer = null;
}
```

Теперь мы имеем половину программы, позволяющей осуществить голосовое соединение. Добавьте к вашему приложению код, приведенный в листинге 20.2.

Листинг 20.2. Присоединение к сеансу голосовой связи.

```
private void ConnectVoice()
{
    // Now create a client to connect
    voiceClient = new Voice.Client(connection) ;
    // Fill in description object for device configuration
    Voice.SoundDeviceConfig soundConfig = new Voice.SoundDeviceConfig(),
    soundConfig.Flags = Voice.SoundConfigFlags.AutoSelect;
    soundConfig.GuidPlaybackDevice = DSoundHelper.DefaultPlaybackDevice;
    soundConfig.GuidCaptureDevice = DSoundHelper.DefaultCaptureDevice;
    soundConfig.Window = this;
    // Fill in description object for client configuration
    Voice.ClientConfig clientConfig = new Voice.ClientConfig();
    clientConfig.Flags = Voice.ClientConfigFlags.AutoVoiceActivated |
    Voice.ClientConfigFlags.AutoRecordVolume;
    clientConfig.RecordVolume = (int) Voice.RecordVolume.Last;
    clientConfig.PlaybackVolume = (int) Voice.PlaybackVolume.Default;
    clientConfig.Threshold = Voice.Threshold.Unused;
    clientConfig.BufferQuality = Voice.BufferQuality.Default;
    ClientConfig.BufferAggressiveness = Voice.BufferAggressiveness.Default;
```

```
    // Connect to the voice session
    voiceClient.Connect(soundConfig, clientConfig, Voice.VoiceFlags.Sync);
    voiceClient.TransmitTargets = new int[] (
        (int)Voice.PlayerId.AllPlayers 1;
}
```

Сначала мы создаем объект voice client и указываем объект DirectPlay, который будет использоваться для передачи голосовых данных. Затем, прежде чем присоединиться к сеансу, мы должны установить конфигурацию для обеих звуковых карт. Для этого используется структура SoundDeviceConfig, сообщающая DirectPlay об устройствах, которые могут использоваться для голосовой связи. В нашем случае мы автоматически выбираем микрофон и заданные по умолчанию устройства захвата и воспроизведения звука (в главе DirectSound мы использовали совместный доступ к этим устройствам, и здесь мы оставим это без изменения).

Далее с помощью структуры ClientConfig мы определяем параметры клиента. Флажки используются таким образом, чтобы максимальное качество звука устанавливалось автоматически и передача начиналась сразу, как только будет получен звук от микрофона. При использовании флажка AutoVoiceActivated необходимо установить пороговое значение в положение «unused». Если же осуществляется «ручной» запуск передачи звука, пороговое значение должно быть установлено на минимальный уровень.

Теперь мы устанавливаем уровень звука по умолчанию, затем соединяемся с сетью (точнее с хостом), для простоты используя при этом флажок синхронности (чтобы не записывать дополнительный обработчик). Присоединившись к сеансу, мы определяем объекты для передачи голосовых сообщений.

И наконец, нам осталось сделать два добавления. Во-первых, необходимо добавить процедуру Dispose, чтобы освободить объект и выйти из соединения (опять используя синхронный флажок):

```
if (voiceClient != null)
{
    voiceClient.Disconnect(Voice.VoiceFlags.Sync);
    voiceClient.Dispose();
    voiceClient = null;
}
```

Во-вторых, мы должны предусмотреть вызов метода ConnectVoice и для сервера, и для клиента: в конце метода StartSession на сервере, чтобы позволить главному компьютеру создание клиента voice client, и в процедуре OnConnectComplete после успешной установки соединения:

```
if (e.Message.ResultCode == ResultCode.Success)
{
    this.BeginInvoke(new AddTextCallback(AddText),
        new object[] { "Connect Success."});
    connected = true;
    this.BeginInvoke(new EnableCallback(EnableSendDataButton),
        new object[] { true } );
    ConnectVoice();
}
```

ПРОВЕРКА УСТАНОВКИ ЗВУКА

Если вы не использовали мастер установки, вызов Connect может выдать ошибку RunSetupException. В этом случае можно «захватить» это исключение и произвести установку звука, используя следующий код:

```
Voice.Test t = new Voice.Test();
t.CheckAudioSetup();
```

После установки можно попробовать подключиться еще раз. Данная установка выполняется один раз.

Краткие выводы

В этой главе мы рассмотрели различные особенности DirectPlay, включая следующие разделы.

- **Модели событий и обработчики.**
- **Пропускная способность, трафик.**
- **Очередность отправки данных.**
- **Лобби-приложения.**
- **Использование голосовой связи в приложении.**

В нашей заключительной главе мы обобщим накопленную к этому моменту информацию.

Глава 21. Методы достижения максимального быстродействия

В данной главе будут охвачены следующие темы.

- Операции преобразования типов: boxing и unboxing.
- Модель события и ее недостатки.
- Повышение эффективности.

Преобразование типов в объекты

При разработке различных программ вопрос быстродействия остается одним из самых основных. Ниже мы рассмотрим реальные примеры, позволяющие увеличить быстродействие приложений.

Пример Billboard (Доска Объявлений), который поставляется с DirectX SDK, имеет различные версии: неуправляемую версию, написанную на C++, а также две управляемые версии, одну, написанная на C#, другую, разработанную в среде VB.NET Поскольку каждый пример из DirectX SDK учитывает частоту смены кадров, можно легко определить, какое из приложений выполняется быстрее.

Сравнивая пример Billboard, написанный на C++, с образцом, написанном на C#, нужно обратить внимание на то, что скорость выполнения C# приложения составляет примерно 60% от скорости образца на C++. Принимая во внимание использование управляемого и неуправляемого кода в различных версиях следует выяснить, с чем связано это «замедление».

Термин «boxing» для .NET Runtime подразумевает преобразование типов в некий объект. Соответственно, процесс обратного преобразования называется «unboxing». Для выполнения этих операций среде .NET Runtime необходимо выделить часть динамической памяти, достаточную для размещения данных, и затем скопировать данные из стека (где хранится значение) в созданную область динамической памяти.

Теперь, используя управляемую версию примера Billboard, можно обратить внимание на то, что каждая отображаемая ветвь имеет соответствующую структуру, содержащую необходимую для рисования информацию. Поскольку структура смешана, но должна отображаться в соответствующем порядке, выполним следующую операцию:

```
trees.Sort (new TreeSortClass());
```

Данный класс имеет встроенную процедуру сравнения IComparer, которая принимает в качестве входных параметров два объекта. Прежде чем она будет передана в метод сравнения, структура используемых объектов должна быть преобразована с помощью метода boxing, а после выполнения процедуры сравнения — преобразована обратно (unboxing).

Метод Sort будет вызываться приблизительно 4300 раз за кадр. При каждом вызове будут выполняться две операции преобразования boxing и, соответственно, две обратных операции. Сама структура определена следующим образом:

```
public struct Tree
{
   public CustomVertex.PositionColoredTextured v0, v1, v2, v3;
   public Vector3 position;
   public int treeTexturelndex;
   public int offsetlndex;
};
```

Можно рассчитать объем памяти, который потребуется для размещения данных в процессе выполнения операции сортировки (исходя из размера структуры равного 116-ти байтам). Для этого необходимо умножить величину объекта (116 байт) на число объектов (2) и на количество вызовов для одного кадра (4300), получается огромное значение — 997,600 байтов на каждый кадр. И это касается только операции размещения.

После того как данные скопированы, и операция сравнения проведена, они должны пройти процедуру unboxing. Это подразумевает те же действия по выделению памяти и копированию данных, только на этот раз для стека. В итоге, данные для каждого кадра (1,995,200 байт) распределяются между стеком и динамической памятью с поочередным копированием, причем размер распределенных областей небольшой (116 байт), а их количество составляет немалое значение. Теперь, по крайней мере, понятна причина замедления выполнения программ на C++. Использование среды .NET Runtime дает огромное преимущество в быстродействии и гибкости, что должно послужить причиной для перехода к программированию в этой среде.

Побочные эффекты моделей событий

Допустим, имеется знакомый сценарий, характерный для Управляемого Direct3D. Мы уже достаточно знакомы с работой обработчиков событий в приложениях Управляемого DirectX, и можем предположить, что многие операции, которые выполняют обработчики, расходуют немало ресурсов, если использовать их не должным образом.

Во-первых, это касается распределения памяти. Например, каждый раз при создании вершинного буфера, помимо обычного резервирования памяти, выделяется дополнительная память для делегатов, необходимых для обработки событий. Обработчики событий непосредственно связаны с этими объектами во время работы приложения. Рассмотрим типичный пример:

```
public void WasteVideoMemory()
{
    VertexBuffer waste = new VertexBuffer(device, 5000,
        0, VertexFormats.None, Pool.Default);
}
```

Этот код выглядит «мертвым». Конечно, вершинный буфер создается, но он никогда не используется, тем не менее, программа «сборщика мусора» все равно обработает данный буфер.

После создания вершинного буфера отслеживаются события DeviceLost, DeviceReset и Disposing. Обработчики этих событий связывают устройство и вершинный буфер, и, если буфер нигде не используется, «сборщик мусора» не заметит его.

Если бы подобный метод вызывался многократно, мы бы постепенно исчерпали весь запас свободной видеопамяти (поскольку объект был создан в заданном по умолчанию пуле памяти).

Другой побочный эффект подобного поведения — время перезагрузки «shutdown time».

Нам написало несколько человек, которые сообщили об ошибках в Managed Runtime, связанных с блокировкой приложения при выходе из системы или перезагрузке. Можно предположить, что сценарий был следующий:

```
device.SetRenderTarget(0, mySwapChain.GetBackBuffer(0,
BackBufferType.Mono));
// Render some stuff
```

Этот код выполняет примерно те же самые операции, которые мы описывали в предыдущем примере. Создается новый объект (в данном случае поверхность), который никогда не используется. Таким образом, в результате многократного выполнения этой строки создаются тысячи «потерянных» поверхностей. При закрытии приложения устройство пытается освободить все эти поверхности (функция dispose) и естественно «зависает».

Есть два пути для решения этой проблемы. Первый, самый простой, — включение директивы using для объектов, которые будут использоваться в ближайшее время. Соответствующий код может иметь вид:

```
using(Surface backBuffer = mySwapChain.GetBackBuffer(0,
BackBufferType.Mono))
{
    device.SetRenderTarget(0, backBuffer);
    // Render some stuff.
}
```

ИСПОЛЬЗОВАНИЕ ДИРЕКТИВЫ USING

Эта директива автоматически размещает создаваемый объект. Компилятор C# разобьет директиву на код, подобный этому:

```
Surface backBuffer;
try
{
    backBuffer = mySwapChain.GetBackBuffer(0,
    BackBufferType.Mono);
    device.SetRenderTarget(0, backBuffer);
    // Render some stuff
}
finally
{
    if (backBuffer != null)
        backBuffer. Dispose();
}
```

Использование директивы using дает существенный выигрыш в быстродействии, особенно для случаев многократного распределения небольших объемов данных в памяти (см. выше). Однако, это не решает проблему выделения дополнительной памяти для процедур обработки событий. Как правило, события и обработчики событий, содержащиеся в Управляемом DirectX, позволяют вам, как разработчику, не заботиться о размерах и времени существования объекта. Тем не менее, в зависимости от требований и задач приложения, иногда весьма полезно управлять свойствами объектов самостоятельно.

В версии SDK Update Управляемого DirectX 9 мы имеем возможность отключать обработчик событий, если мы знаем заранее, что он нам не понадобится:

```
Device.IsUsingEventHandlers = false;
device = new Device(...);
```

Значение «true» установлено по умолчанию, и мы можем изменить его в любое время. Устанавив «false» еще до создания устройства, мы полностью отключаем обработку событий в приложении.

ОТКЛЮЧЕНИЕ ОБРАБОТКИ СОБЫТИЙ

Использование этой особенности выключит автоматическое отслеживание событий. Использование внешнего управления обработчиками потребует некоторого опыта и сноровки, поэтому при написании программ тщательно проверяйте ваши действия.

Итак, теперь в большинстве случаев мы можем использовать обработчики в нашем приложении, при необходимости отключая некоторые из них. Рассмотрим следующий код:

```
// Device is currently using event handlers
Device.IsUsingEventHandlers = false;
using(Surface backBuffer = mySwapChain.GetBackBuffer(0,
BackBufferType.Mono))
{
   device.SetRenderTarget(0, backBuffer);
   // Render some stuff.
}
// Allow device events to be hooked again
Device.IsUsingEventHandlers = true;
```

Данная процедура отключает обработчики событий на то время, пока создается временная поверхность, и после этого включает их опять. В зависимости от наших требований, мы можем выключить обработку событий полностью или частично, добиваясь таким образом максимальной эффективности.

ИСПОЛЬЗОВАНИЕ УСТРОЙСТВА «THREADING DEVICE»

Так как существует возможность отключения обработки событий, мы можем использовать в устройстве мультипоточный режим, включив соответствующий флажок в используемой для создания устройства структуре параметров. Если мы собираемся управлять обработкой событий «вручную», это можно делать в общем потоке:

```
presentParams.ForceNoMultiThreadedFlag = true
```

Эффективность методов

Знание эффективности каждого из используемых методов в отдельности может существенно помочь при написании быстрого и эффективного приложения.

В Управляемом DirectX имеются случаи, когда, казалось бы, «безобидный» код может вызвать значительные проблемы. Например, любой метод, принимающий в качестве параметра строку, должен разместить в памяти неуправляемый строковый тип и скопировать туда данный параметр.

Возьмем случай, где мы используем язык шейдеров HLSL для отображения нашей сцены, и будем переключать используемые для каждого кадра техники:

```
myEffeet.Technique = "TransformWorldSpace";
//Render some stuff
myEffeet.Technique = "TransformAndAddGlow";
// Render some other stuff
```

Каждый раз, когда мы переключаем методику, происходит перераспределение памяти и копирование названия методики. Решить данную проблему можно с помощью кэширования возвращаемых указателей, например:

```
// On device creations, cache technique handles
EffectHandle handlel = myEffect.GetTechnique("TransformWorldSpace");
EffectHandle handle2 = myEffeet.GetTechnique("TransformAndAddGlow");
// Later on, for every render
myEffeet.Technique = handlel;
//Render some stuff
myEffeet.Technique = handle2;
//Render some stuff
```

Краткие выводы

В этой главе мы рассмотрели следующие вопросы.

- **Операции преобразования типов boxing и unboxing.**
- **Использование модели события и ее недостатки.**
- **Эффективность используемых методов.**

ЧАСТЬ VII

ПРИЛОЖЕНИЯ

Приложение А. Использование сборок диагностики

Предположим, что вы только что завершили написание игры и готовите ее к выходу в свет. Несмотря на положительные тестовые результаты вашей испытательной группы, у вас все равно не было возможности проверить это приложение на всех системах без исключения.

Допустим, что приблизительно 5 % всех пользователей, купивших игру, не смогут запустить ее по каким-либо причинам. У вас есть предположения, что это связано с типом используемой видеокарты. Попробуем проверить это, используя пространство имен Diagnostics.

В этом приложении мы рассмотрим следующие вопросы.

Перечисление всех диагностических опций и вывод сообщений о них.

Проверка отдельных пунктов.

Перечисление всех опций в системе

Пространство имен Diagnostics включает в себя те же опции, что и инструмент DxDiag. Используя его, можно узнать практически все о вашей системе

Данные диагностики размещаются иерархически. Можно использовать рекурсивную функцию для перечисления всех возможных опций и объектов, находящихся в указанном корневом контейнере. Данная функция приведена в листинге АЛ и включена в образец DirectX SDK DxDiagOutput.

Листинг A.1. Выводимые данные диагностики.

```
static void OutputDiagData(string parent, Container root)
{
   try
   {
      foreach (PropertyData pd in root.Properties)
      {
         // Just display the data
         Console.WriteLine("{0}.{1} = {2}", parent, pd.Name, pd.Data);
      }
}
catch


try
{
```

```
        foreach (ContainerData cd in root.Containers)
        {
            // Recurse all the internal nodes
            if (parent == null)
                OutputDiagData(cd.Name, cd.Container);
            else
                OutputDiagData(parent + V + cd.Name, cd.Container);
        }
    }
    catch
    {
    }
    // We are done with this container, we can dispose it.
    root.Dispose();
}
```

Здесь выполняется поиск и отображение всех имеющихся свойств и опций, а также соответствующих им значений. В зависимости от свойства значения будут либо строчными, либо булевыми, либо целочисленными. При необходимости дополнительную информацию можно получить, используя метод pd.Data.GetType().

После того как мы получили список свойств данного контейнера, необходимо проверить, содержит ли он дочерние контейнеры. Если да, просматриваем каждый из них.

Для начала необходимо создать хотя бы первый контейнер и вызвать указанный метод. Объект контейнера содержит только один конструктор, имеющий в качестве параметра одну логическую переменную. Это значение используется, чтобы определить, должна ли диагностика включать в себя информацию об аппаратных средствах WHQL (Windows Hardware Quality Labs).

Получение этой информации может занять продолжительное время, лоэтому если в данной операции нет необходимости, ее лучше пропустить. Образец DirectX SDK использует точку входа в программу, приведенную в листинге А.2.

Листинг А.2. Запуск приложения.

```
static void Main(string[] args)
{
    try
    {
        // Just start our recursive loop with our root container. Don't worry
        // about checking Whql
        OutputDiagData(null, new Container(false));
    }
```

```
    catch
    {
        // Something bad happened
    }
}
```

Таким образом, сформировав опции и свойства диагностики в вашем приложении, можно обнаружить искомый сбой в работе программы.

Проверка отдельных пунктов

Как известно, полное перечисление всех опций и свойств (как в DxDiag, так и в DirectX SDK) может занимать немало времени. Допустим, мы хотели бы получить информацию относительно небольшого числа пунк-. тов.

Ниже приведен пример кода, позволяющего получить определенную информацию о версии DirectX, которая использовалась в вашей системе:

```
Container parent = new Container(false);
Container child = parent.GetContainer("DxDiag_SystemInfo").Container;
int dxVersionMajor = (int)child.GetProperty("dwDirectXVersionMajor").Data;
int dxVersionMinor = (int)child.GetPropertyCdwDirectXVersionMinor") .Data;
string dxVersionLetter =
 (string)child.GetProperty("szDirectXVersionLetter").Data;
Console.WriteLinePDX
Version:(01.(1)(21",dxVersionMajor,dxVersionMinor,dxVersionLetter);
```

Мы создаем корневой (без информации WHQL) контейнер. Затем, после образования дочернего контейнера DxDiag Systemlnfo, получаем три различных свойства версии DirectX: главный и вспомогательный номера версии, а также символ, связанный с этой версией (известна, например, версия DX8.1B).

Префиксы в именах свойств могут определять заданный для объекта по умолчанию тип данных. Эти имена соответствуют Венгерскому примечанию для разработчиков программ на С. Пункты с префиксом «sz» означают строки, пункты с префиксом «b» — логические значения. Все другие префиксы относятся к целочисленным значениям.

Использование пространства имен Diagnostics может оказать существенную помощь при наладке и проверке вашего приложения. А возможность обратной связи с пользователями позволяет отследить работу приложения на самых разнообразных системах и видеокартах.

Как говорится, — «Лучше планировать проблему до, чем ждать и сожалеть после».

Приложение В. Воспроизведение музыки и видео

Как мы уже писали в начале книги, в окончательной версии Управляемого DirectX SDK API мы исключили компоненты DirectMusic и DirectShow. Вместо этого мы включили пространство имен AudioVideoPlayback, позволяющее запускать видеоролики и музыкальные файлы (например, файлы mp3s или wma). В этом приложении мы обсудим следующие вопросы.

- Простое воспроизведение звукового файла.
- Простое проигрывание видео файла.
  Проигрывание видео файла в отдельном окне.
  Использование видео файла в качестве текстуры в 3D приложении.

Воспроизведение звукового файла

Прежде чем начать работу со звуковыми и видео файлами, необходимо добавить в приложение соответствующие ссылки на пространство имен AudioVideoPlayback, а также директиву using.

При создании пространства имен AudioVideoPlayback нашей главной задачей ставилась простота в использовании. В итоге, мы можем запустить воспроизведение музыкального файла с помощью одной короткой строки, например, код для запуска звукового файла «piano.mp3» выглядит следующим образом:

```
Audio someAudio = new Audio("piano.mp3", true);
```

Имеются два основных класса, которые включены в пространство имен AudioVideoPlayback: класс Video и класс Audio. Эти названия довольно очевидны. Конструктор для каждого класса имеет два варианта. Первый параметр для каждого из вариантов представляет собой имя файла, который мы хотим запустить. Другой параметр содержит булево значение, которое определяет автоматическое воспроизведение этого файла, в нашем случае «true» (по умолчанию «false», то есть файл не будет проигрываться без соответствующей команды). Таким образом, мы создали звуковой объект, который будет запускаться автоматически.

Конструкторы — не единственный способ создавать (или загружать) эти файлы. Наиболее близким является статический метод FromFile, который принимает те же самые параметры и возвращает новый экземпляр класса. Имеется также статический метод FromUri, который ведет себя подобно методу FromFile, за исключением того, что он загружает данные с web-узла (или любого корректного URL). При этом воспроизведение при использовании URL может начаться еще до того, как файл загрузится полностью.

Существуют также методы Open и OpenUrl с теми же входными параметрами. Они заменяют данные в уже созданном объекте на данные, полученные из нового файла или web-узла.

Воспроизведение видео файла в отдельном окне

В предыдущем примере для показа простоты работы мы использовали звуковой файл. Допустим, мы хотим сделать то же самое с видео файлом (например, вместо звукового файла запустить видео ролик «butterfly.mpg»).

Для этого достаточно использовать следующий код:

```
Video someVideo = new Video("butterfly.mpg", true);
```

Обратите внимание, что в этом случае воспроизведение будет осуществляться в новом окне.

Немного изменив код, мы можем воспроизводить ролик в уже имеющемся окне. Для этого необходимо отключить автоматическое воспроизведение (убрав значение «true» в первой строке) и добавить еще две дополнительные строки:

```
Video someVideo = new Video("butterfly.mpg");
someVideo.Owner = this;
someVideo.Play();
```

После отмены режима автоматического воспроизведения мы устанавливаем «владельца» файла, который может быть любым объектом Windows-форм. Этим «владельцем» может быть непосредственно наша основная или ее дочерняя форма (например, picture box). После всех этих действий мы, наконец, запускаем видео. Обратите внимание, что теперь ролик прокручивается внутри окна, которое мы можем определить в свойствах «владельца».

Мы можем управлять звуком воспроизводимого видео файла (если он содержит звуковые данные) с помощью свойства Audio, отвечающего за соответствующий нашему видео звуковой объект, с которым мы можем оперировать, как с обычным звуковым объектом.

Использование видео файла в качестве текстуры

В этом разделе мы покажем, как можно использовать видео файл в качестве 3О-текстуры в нашем 3О-приложении. Многие разработчики

используют в своих играх различные ролики из кинофильмов в качестве фона или деталей игры, и пространство имен AudioVideoPlayback достаточно хорошо справляется с этой задачей.

В качестве наиболее простого случая возьмем фрагмент кинофильма и запустим его в полноэкранном режиме. Для начала просто загрузите видео файл в объект Video, установите опцию Fullscreen в значение «true» и запустите ролик. Пока все достаточно просто.

Рассмотрим более сложный сценарий с использованием объектов текстурирования. Начнем по порядку. У нас имеется событие TextureReadyToRender, которое мы хотели бы отслеживать. Данное событие будет зафиксировано каждый раз, когда в видео файле будет появляться новый кадр, который мы хотим отобразить в качестве текстуры.

Параметр TextureRenderEventArgs процедуры обработчика события содержит вновь созданную текстуру, которая готова для отображения. Поскольку это событие может быть отслежено из различных потоков, необходимо убедиться в том, что мы не используем множественные потоки, одновременно обращающиеся к переменным (никто не знает, чем это может закончиться).

Теперь все, что необходимо для запуска игры, — это вызвать для видео объекта метод RenderToTexture.

Пример, который поставляется с DirectX SDK, отобразит полную сцену игры с той же самой скоростью смены кадров, как и в кинофильме. Во многих случаях это вполне приемлемо для разработчиков, поскольку видео-текстура явление довольно редкое. Но что необходимо сделать, если мы хотим использовать текстуру в нашем приложении, работающем в режиме реального времени.

Вначале мы должны выяснить, где создана эта видео текстура: в системном пуле памяти или в заданном по умолчанию пуле. Это можно сделать, проверив описание первого уровня текстуры.

Если текстура была создана в заданном по умолчанию пуле памяти (наиболее общий случай), для создания текстуры в устройстве DirectSD необходимо использовать метод GetRenderTargetData и поверхность, созданную нами в системном пуле памяти. Поскольку мы не можем отображать текстуры, созданные в системном пуле памяти, необходимо создать используемую для рендеринга текстуру в заданном по умолчанию пуле памяти, например:

```
SurfaceDescription ds = e.Texture.GetLeyelDescription(0);
if (ds.Pool == Pool.Default)
{
   systemSurface = device.CreateOffscreenPlainSurfacefds.Width, ds.Height,
      ds.Format, Pool.SystemMemory);
}
```

```
texture = new Texture(device, ds.Width, ds.Height,
    1, Usage.Dynamic, ds.Format, Pool.Default);
```

Отсюда видно, что если видео текстура была создана в заданном по умолчанию пуле памяти, приложение создаст за кадром поверхность для отображения на нее нашей текстуры, а также текстуру в заданном по умолчанию пуле памяти, которую мы можем использовать для рендеринга нашей сцены. После того как это будет сделано, необходимо применить два метода, позволяющие устройству передать данные от одной текстуры к другой:

```
using(Surface videoSurface = e.Texture.GetSurfaceLevel(0))
{
   using(Surface textureSurface = texture.GetSurfaceLevel(0))
   {
      device.GetRenderTargetData(videoSurface, systemSurface);
      device.UpdateSurface(systemSurface, textureSurface);
   }
}
```

Как вы можете видеть, мы получаем данные рендеринга от нашей видео текстуры и перемещаем их на нашу временную поверхность, созданную в системной памяти. Оттуда мы перемещаем эту поверхность на нашу реальную текстуру. Только затем мы можем использовать текстуру, созданную в заданном по умолчанию пуле памяти, в качестве текстуры для рендеринга нашей сцены.

Поскольку в нашем случае текстура доступна для нескольких потоков, мы должны гарантировать (используя семантику блокировки), что только один поток одновременно имеет доступ к этой текстуре.

В завершение следует отметить, что класс AudioVideoPlayback разработан не как подобие DirectShow. Этот класс не предназначен для некоторых опций, таких как, например, захват видеоизображений. Описанные объекты разработаны для наиболее общих и простых случаев загрузки и воспроизведения звуковых и видео данных.

Описанные классы эффективны и удобны, но при этом они не могут рассматриваться как полноценная альтернатива DirectShow API.

На данный момент мы не имеем каких-либо запланированных обновлений для этого пространства имен Управляемого DirectX.