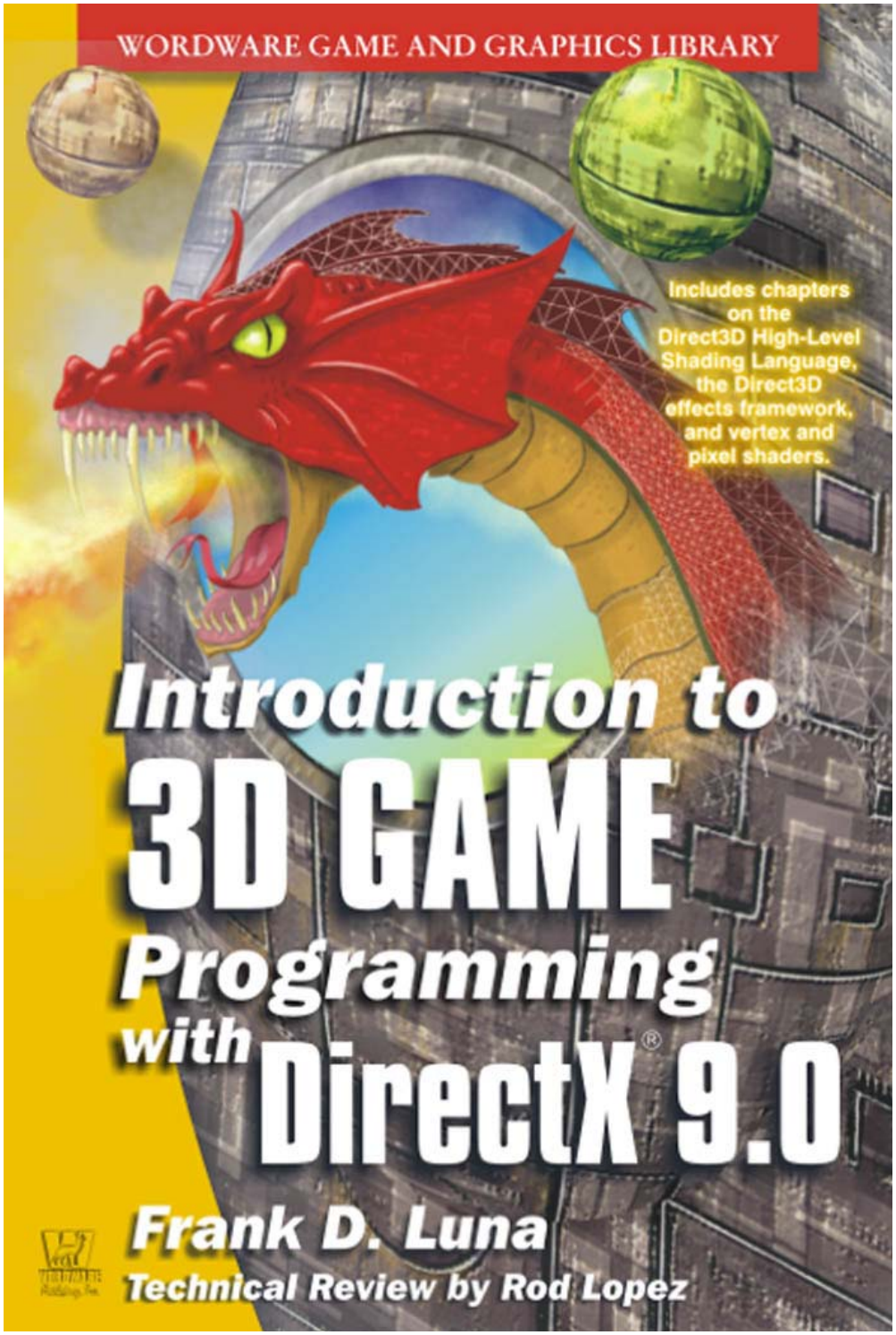
WORDWARE GAME AND GRAPHICS LIBRARY
Includes chapters on the Direct3D High-Level Shading Language, the Direct3D effects framework, and vertex and pixel shaders.
Introduction to
3D GAME
Programming
with DirectX® 9.0
Frank D. Luna
Technical Review by Rod Lopez

СЕТЕВАЯ БИБЛИОТЕКА

Фрэнк Д. Луна

ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ ТРЕХМЕРНЫХ ИГР С DirectX 9.0

NetLib 2006

Фрэнк Д. Луна

Введение в программирование трехмерных игр с DirectX 9.0

Wordware Publishing, 2003
ISBN: 1-55622-922-4

Аннотация

Книга «Введение в программирование трехмерных игр с DirectX 9.0» представляет собой вводный курс программирования интерактивной трехмерной компьютерной графики с использованием DirectX 9.0, в котором основное внимание уделяется разработке игр. Книга начинается с исследования необходимых математических инструментов и базовых концепций трехмерной графики. Другие темы охватывают как выполнение в Direct3D базовых операций, таких как рисование графических примитивов, освещение, наложение текстур, альфа-смешивание и работу с трафаретами, так и использование Direct3D для реализации техник, необходимых в играх. Главы посвященные вершинным и пиксельным шейдерам включают описание каркасов эффектов и нового высокоуровневого языка программирования шейдеров (HLSL).

Оформление и подготовка издания	Сетевое издательство NetLib, 2006
Сайт	*netlib.narod.ru*
E-mail	*netlib@mail.ru*

Оглавление

Моим родителям, Фрэнку и Кэтрин.

Благодарности

Я хочу поблагодарить моего технического редактора, Рода Лопеза (Rod Lopez) за время, потраченное на эту книгу, за проявленную при этом скрупулезность и за предложенные улучшения. Я также хочу поблагодарить Джима Лейтермана (Jim Leiterman) (автора книги «Vector Game Math Processor» выпущенной издательством Wordware Publishing) и Хэйли Лиунга (Hanley Leung) (программиста Kush Games) за рецензирование отдельных частей книги. Затем я хочу сказать спасибо Адаму Хаулту (Adam Hault) и Гари Симмонсу (Gary Simmons), ведущим учебный курс по BSP/PVS на сайте www.gameinstitute.com, за предоставленную помощь. Кроме того, я должен поблагодарить Вильямя Чина (William Chin), который помог мне много лет назад. И, наконец, я благодарю персонал издательства Wordware Publishing, в том числе Джим Хилл (Jim Hill), Вес Беквит (Wes Beckwith), Бет Кохлер (Beth Kohler), Хитер Хилл (Heather Hill), Дениз МакЭвой (Denise McEvoy) и Алана МакКаллера (Alan McCuller).

Введение

Эта книга представляет собой введение в программирование интерактивной трехмерной компьютерной графики с использованием DirectX 9.0, в котором основное ударение делается на разработку игр. Прочитав ее, вы изучите основы Direct3D, после чего сможете перейти к изучению и применению более сложных технологий. Раз вы держите в руках эту книгу, у вас есть начальное представление о том, что представляет собой DirectX. С точки зрения разработчика DirectX — это набор API (интерфейсов программирования приложений) для разработки мультимедийных приложений на платформе Windows. В этой книге мы сосредоточимся на ограниченном подмножестве DirectX, называемом Direct3D. Как видно из названия, Direct3D — это API, используемый для разработки приложений с трехмерной графикой.

Книга разделена на четыре части. В первой части исследуются математические инструменты, применяемые в остальных частях книги. Во второй части объясняются базовые технологии трехмерной графики, такие как освещение, текстурирование, альфа-смешивание и работа с трафаретами. Третья часть посвящена использованию Direct3D для реализации различных более интересных техник и приложений, таких как выбор объектов, визуализация ландшафтов, системы частиц, настраиваемая виртуальная камера, а также загрузка и визуализация трехмерных моделей (X-файлов). Темой четвертой части являются вершинные и пиксельные шейдеры, включая каркасы эффектов и новый (появившийся в DirectX 9.0) высокоуровневый язык шейдеров. Настоящее и будущее трехмерных игр неразрывно связано с использованием шейдеров, и, посвящая им целую часть, мы хотели получить книгу, соответствующую современному состоянию дел в программировании графики.

Новичкам лучше всего читать эту книгу с начала и до конца. Главы упорядочены таким образом, что сложность материала постепенно возрастает с каждой новой главой. Предложенный путь позволит избежать скачкообразного увеличения сложности, которое может оставить читателя в недоумении. Часто в отдельной главе мы используем техники и концепции, объясненные ранее. Поэтому, перед тем, как продолжить чтение, необходимо тщательно изучить материал текущей главы. Опытные читатели могут выбирать те главы, которые им интересны.

Наконец вы можете задаться вопросом — какие игры вы сможете разрабатывать после прочтения этой книги. Чтобы ответить на него лучше всего пролистать книгу и взглянуть на типы разрабатываемых приложений. Это поможет вам представить типы игр, которые можно разработать, основываясь на рассматриваемых в книге методах и вашей собственной изобретательности.

Предпосылки

Эта книга разрабатывалась таким образом, чтобы служить учебником начального уровня. Однако это не означает, что она будет простой для людей, не имеющих опыта программирования. Ожидается, что читатель хорошо знает алгебру, тригонометрию, свою среду разработки (т.е. Visual Studio), C++ и фундаментальные структуры данных, такие как массивы и списки. Опыт программирования для Windows тоже будет полезен, но он не обязателен — получить начальные сведения о программировании для Windows можно в Приложении A.

Необходимые средства разработки

В качестве языка программирования для всех представленных в книге примеров программ используется C++. Процитируем документацию к DirectX: «DirectX 9.0 поддерживается только Microsoft Visual C++ 6.0 и более поздними версиями». Следовательно, для того, чтобы писать использующие DirectX 9.0 приложения на C++, вам необходим либо Visual C++ (VC++) 6.0 либо VC++ 7.0 (.NET).

ПРИМЕЧАНИЕ Код примеров для этой книги компилировался и строился с использованием VC++ 7.0. Его большая часть должна компилироваться и строиться также и в VC++ 6.0, но при этом следует помнить об одном существенном отличии. Приведенный ниже фрагмент кода успешно компилируется в VC++ 7.0, поскольку переменная **cnt** считается локальной для цикла **for**.

```
int main() {
   for(int cnt = 0; cnt < 10; cnt++) {
      std::cout << "hello" << std::endl;
   }
   for(int cnt = 0; cnt < 10; cnt++) {
      std::cout << "hello" << std::endl;
   }
   return 0;
}
```

Однако в VC++ 6.0 этот код компилироваться не будет. Вы получите сообщение об ошибке ***C2374:** 'cnt': redefinition; multiple initialization* поскольку в VC++ 6.0 переменная **cnt** не считается локальной для цикла. Поэтому, перенося код в VC++ 6.0, вы должны сделать в нем небольшие изменения, чтобы он компилировался с учетом этого различия.

Рекомендуемое оборудование

Приведенные ниже рекомендации предназначены тем, кто хочет, чтобы примеры программ работали с приемлемой частотой кадров. Любой пример может выполняться с использованием устройства REF, которое осуществляет программную эмуляцию всех функциональных возможностей Direct3D. Поскольку все возможности эмулируются программно, приложения будут работать очень медленно. Более подробно устройство REF будет обсуждаться в главе 1.

Примеры программ из второй части книги очень просты и должны работать на старых бюджетных моделях видеокарт, таких как Riva TNT и аналогичные. Примеры программ из третьей части требуют большего числа геометрических преобразований и используют некоторые новые возможности, такие как точечные спрайты. Для них рекомендуется использовать видеокарты уровня GeForce 2. Примеры программ, приведенные в четвертой части, используют вершинные и пиксельные шейдеры. Поэтому для того, чтобы они выполнялись в реальном времени, вам потребуется видеокарта с поддержкой шейдеров, такая как GeForce 3.

Для кого предназначена эта книга

Создавая эту книгу мы адресовали ее следующим категориям читателей:

- Программистам со средним уровнем знаний C++, которые хотят узнать о программировании трехмерной графики с использованием последней версии Direct3D — Direct3D 9.0.
- Программистам трехмерной графики, обладающим опытом работы с другими API (например, OpenGL), которые хотят узнать о Direct3D 9.0.
- Программистам, имеющим опыт работы с Direct3D, которым требуется соответствующая текущему состоянию дел книга, описывающая последнюю версию Direct3D, включая вершинные и пиксельные шейдеры, высокоуровневый язык шейдеров и каркасы эффектов.

Установка DirectX 9.0

Чтобы писать и выполнять программы, использующие DirectX 9.0, вам необходимо установить на компьютер как библиотеки времени выполнения DirectX 9.0, так и DirectX 9.0 SDK (Software Development Kit). Обратите внимание, что при установке SDK устанавливаются и библиотеки времени выполнения. DirectX SDK можно бесплатно скачать с сайта *http://msdn.microsoft.com/downloads*. Установка достаточно проста, и внимания заслуживает только один момент. В диалоговом окне, изображенном на рис. I.1 необходимо установить переключатель **Debug**.

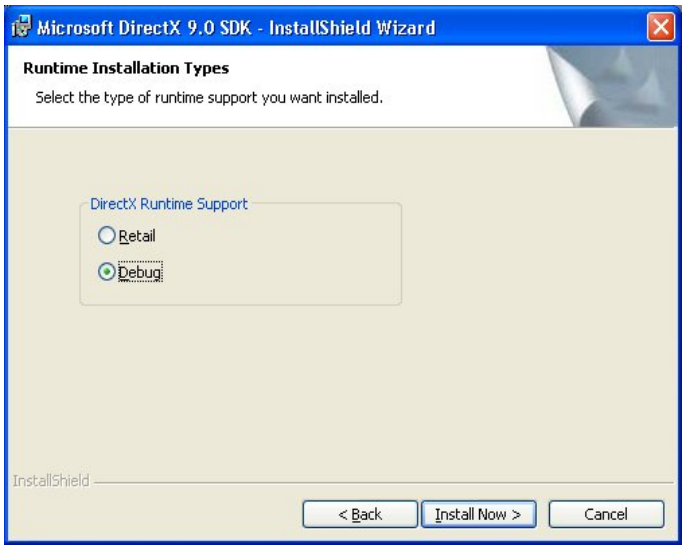

Рис. I.1. *Для разработки приложений при установке DirectX лучше выбрать вариант* ***Debug****, поскольку это упростит отладку*

Если выбран переключатель **Debug**, на компьютер будут установлены как отладочные, так и дистрибутивные версии DLL DirectX, а если выбран переключатель **Retail**, будут установлены только дистрибутивные версии DLL. Для разработки предпочтительнее отладочные версии DLL, поскольку они могут выводить связанную с Direct3D отладочную информацию в окно Visual Studio, когда программа работает в отладочном режиме. Несомненно это очень полезно при отладке приложений, использующих DirectX. На рис. I.2 показана отладочная информация, выводимая когда объект Direct3D не был корректно освобожден.

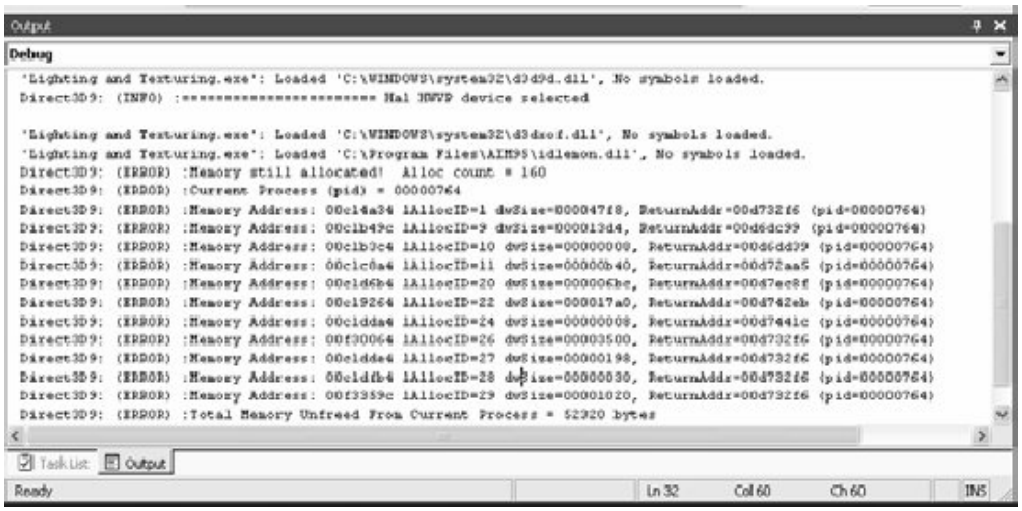

Рис. I.2. *Отладочные сообщения, выводимые если объект Direct3D не был освобожден*

ПРИМЕЧАНИЕ	Помните, что отладочные DLL работают медленнее, чем дистрибутивные. Поэтому при компиляции передаваемого в продажу приложения следует использовать дистрибутивные версии.

Настройка среды разработки

Для написания приложений, использующих DirectX вам следует выбрать тип приложения *Win32 Application*. Кроме того, в VC++ 6.0 и 7.0 вы должны указать пути к каталогам с заголовочными и библиотечными файлами DirectX чтобы VC++ мог найти необходимые файлы. В нашем случае заголовочные и библиотечные файлы DirectX расположены по путям D:\DXSDK\Include и D:\DXSDK\Lib, соответственно.

ПРИМЕЧАНИЕ	Местоположение каталога DirectX DXSDK на вашем компьютере может быть другим. Оно зависит от пути, который вы задали во время установки.

Обычно программа установки DirectX SDK сама добавляет эти пути в VC++ за вас. Однако, если это не произошло, проделайте все вручную, выполнив следующие действия:

В VC++ 6.0 выберите в меню команду **Tools | Options | Directories** и введите пути к заголовочным и библиотечным файлам DirectX, как показано на рис. I.3.

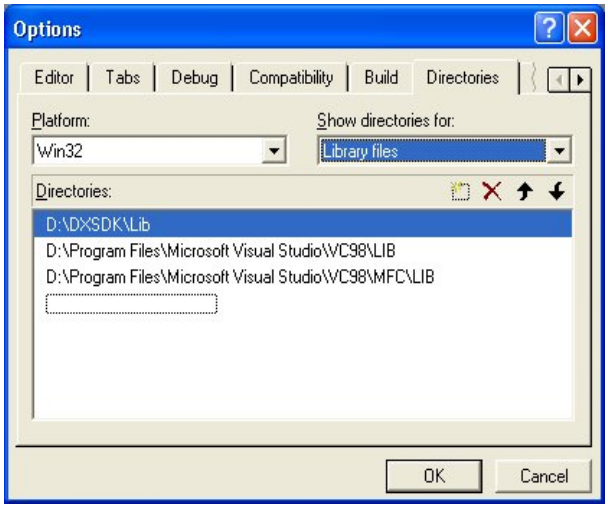

Рис. I.3. *Добавление путей к включаемым и библиотечным файлам DirectX в VC++ 6.0*

В VC++ 7.0 выберите в меню команду **Tools | Options | Projects Folder | VC++ Directories** и введите пути к заголовочным и библиотечным файлам DirectX, как показано на рис. 1.4.

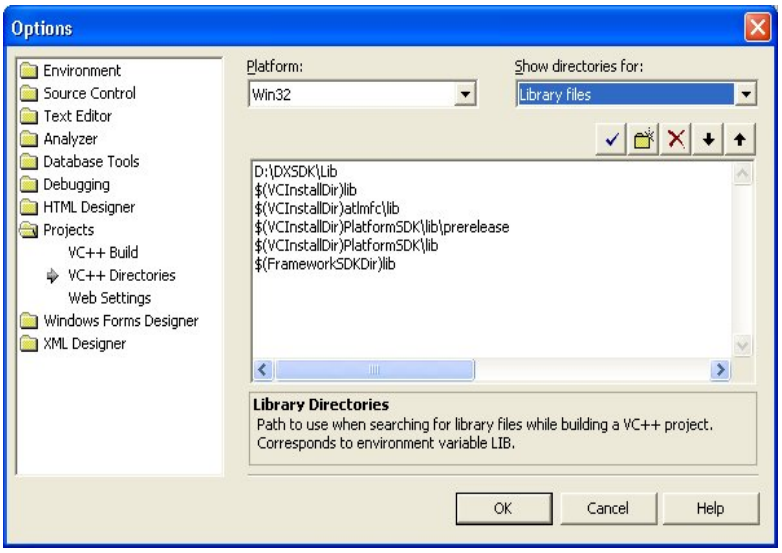

Рис. I.4. *Добавление путей к включаемым и библиотечным файлам DirectX в VC++ 7.0*

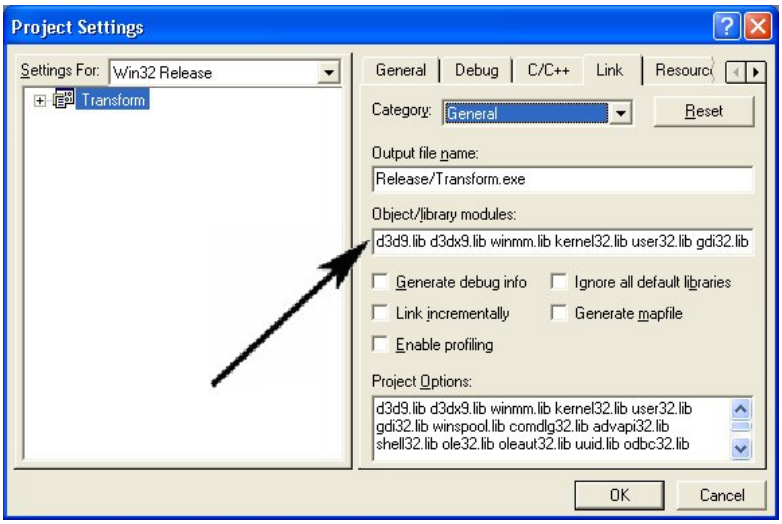

Рис. I.5. *Указание библиотекчных файлов для компоновки с проектом в VC++ 6.0*

Затем, чтобы примеры программ были корректно построены, ваш проект необходимо скомпоновать с библиотечными файлами d3d9.lib, d3dx9.lib и winmm.lib. Обратите внимание, что winmm.lib — это не библиотечный файл DirectX, а библиотечный файл с мультимедийными функциями Windows, и мы будем использовать находящиеся в нем функции работы с таймером.

В VC++ 6.0 чтобы указать библиотечные файлы для компоновки, выберите в меню команду **Project | Settings**, перейдите на вкладку **Link** и введите имена библиотек, как показано на рис. I.5.

В VC++ 7.0 чтобы указать библиотечные файлы для компоновки, выберите в меню команду **Project | Properties | Linker | Input Folder** и введите имена библиотечных файлов, как показано на рис. I.6.

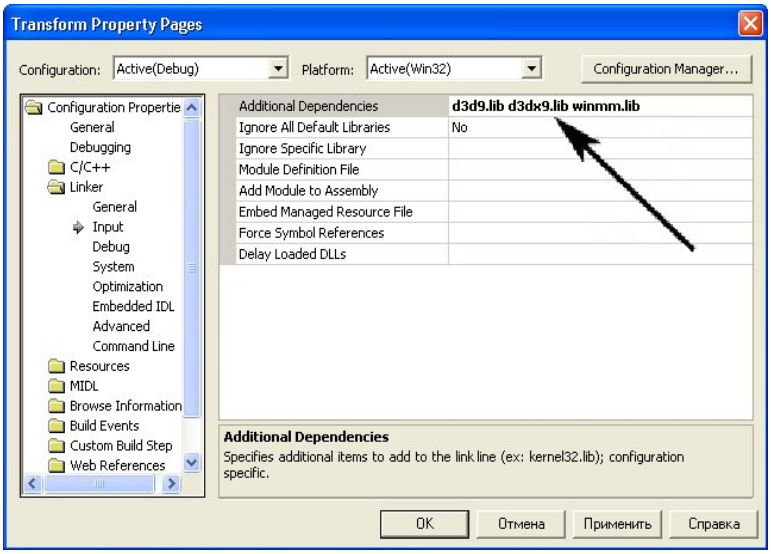

Рис. I.6. *Указание библиотекчных файлов для компоновки с проектом в VC++ 7.0*

Использование библиотеки D3DX

Начиная с версии 7.0, в состав DirectX входит библиотека D3DX (Direct3D Extension). Эта библиотека предоставляет набор функций, классов и интерфейсов, упрощающих выполнение обычных операций, связанных с трехмерной графикой, таких как математические операции, работа с текстурами и изображениями, операции с сетками и операции с шейдерами (например, компиляция и сборка). Можно сказать, что D3DX содержит много возможностей, которые были бы рутинной работой, если бы вам их пришлось выполнять самостоятельно.

На протяжении всей книги мы будем использовать библиотеку D3DX, что позволит нам сосредоточиться на более интересном материале. Например, мы предпочитаем не тратить страницы книги на объяснение того, как загружать изображения различных форматов (bmp, jpeg и т.д.) в интерфейс текстуры Direct3D, когда мы можем сделать это с помощью единственного вызова функции **`D3DXCreateTextureFromFile`**. Другими словами, D3DX делает наш труд более производительным и позволяет сосредотачиваться на более важном материале, а не тратить время на повторное изобретение колеса.

Есть и другие причины для использования D3DX:

- D3DX является библиотекой общего назначения и может использоваться в различных типах приложений, связанных с трехмерной графикой.

- D3DX работает быстро (по крайней мере, настолько быстро, насколько это может делать библиотека общего назначения).
- Другие разработчики используют D3DX. Поэтому, скорее всего, вы столкнетесь с кодом, который использует D3DX. Следовательно, хотите ли вы использовать библиотеку D3DX или нет, вы должны быть знакомы с ней, чтобы суметь разобраться в коде, который использует ее.
- D3DX уже существует и тщательно протестирована. Более того, с каждой новой версией DirectX эта библиотека совершенствуется и в нее добавляются новые возможности.

Использование документации и примеров из DirectX SDK

Direct3D — огромный API и мы даже не надеемся описать все его детали в одной книге. Поэтому, для того, чтобы получать дополнительную информацию, крайне важно научиться использовать документацию DirectX SDK. Чтобы получить доступ к документации по использованию DirectX из C++, откройте файл DirectX9_c.chm из каталога \DXSDK\Doc\DirectX9, где DXSDK — каталог, в который вы установили DirectX.

Документация DirectX охватывает почти все части DirectX API и поэтому очень полезна в качестве справочника. С другой стороны, документации не хватает глубины в изложении деталей, и поэтому она не столь хороша в качестве учебника. Тем не менее с каждой версией DirectX документация становится все лучше и лучше.

Как было сказано, в основном документация используется в качестве справочника. Предположим, вы наткнулись на относящийся к DirectX тип данных или функцию (скажем, функцию **`DSDXMatrixInverse`**), о которой хотите получить больше информации. Вы просто выполняете поиск в алфавитном указателе документации и получаете описание объекта данного типа, или, в нашем случае, функции, как показано на рис. I.7.

ПРИМЕЧАНИЕ В этой книге мы время от времени будем отсылать вас к документации за дополнительными сведениями.

Документация к SDK также содержит несколько учебных примеров, которые расположены по URL *\DirectX9_c.chm::/directx/graphics/programmingguide/tutorials-ndsamplesandtoolsandtips/tutorials/tutorials.htm*. Эти учебные примеры соответствуют некоторым темам, рассматриваемым во второй части книги. Поэтому мы рекомендуем вам изучить эти примеры, когда будете читать соответствующую часть книги, чтобы вы могли получить альтернативные объяснения и альтернативные примеры.

Мы также хотели бы обратить ваше внимание на поставляемые с DirectX SDK примеры программ, работающих с Direct3D. Примеры на C++ находятся в

каталоге \DXSDK\Samples\C++\Direct3D. Каждый пример демонстрирует как реализовать в Direct3D определенный эффект. Эти примеры достаточно сложны для начинающего программиста графики, но закончив читать книгу, вы будете готовы к их изучению. Исследование этих примеров хорошо подходит на роль «следующего шага» после завершения чтения этой книги.

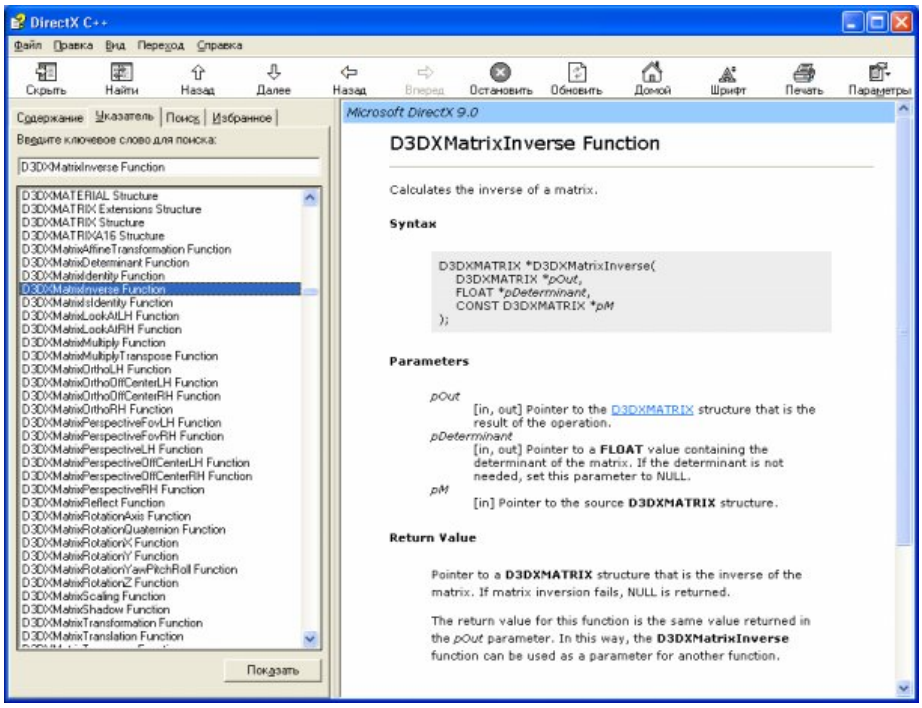

Рис. I.7. *Просмотр документации, относящейся к SDK*

Соглашения об оформлении кода

Соглашения об оформлении кода примеров программ достаточно просты. Отдельного упоминания заслуживают лишь два момента. Во-первых, имена переменных класса начинаются с символа подчеркивания. Например:

```
class C {
   public:
      // ...определение открытого интерфейса
   private:
      float _x; // имя переменной начинается с подчеркивания
      float _y;
      float _z;
};
```

Имена всех глобальных переменных и функций начинаются с заглавной буквы, а имена локальных переменных и методов — со строчной. Мы находим эти приемы полезными для определения области видимости переменной или функции.

Обработка ошибок

В большинстве случаев в примерах программ мы не выполняем никакой обработки ошибок, потому что не хотим отвлекать ваше внимание от более важного кода, который демонстрирует определенную концепцию или технику. Другими словами, мы считаем, что код более ясно демонстрирует концепцию, когда в нем отсутствует проверка ошибок. Помните об этом, когда будете использовать код из примеров в своих собственных проектах. Возможно, вы захотите переработать его, чтобы включить обработку ошибок.

Ясность

Мы хотим подчеркнуть, что примеры программ для этой книги разрабатывались так, чтобы код был легким для восприятия, а не самым быстрым. Поэтому многие примеры программ могут работать неэффективно. Помните об этом, если будете использовать код из примеров в своих собственных проектах, поскольку вы можете захотеть переработать его для повышения эффективности.

Примеры программ и дополнительная поддержка

Веб-сайт для этой книги (*www.moon-labs.com*) играет важную роль, если вы хотите получить от книги максимум возможного. Там вы найдете полный исходный текст всех рассматриваемых в книге примеров. Мы побуждаем читателей изучить соответствующие примеры либо в процессе чтения главы, либо сразу после того, как глава прочитана. Как правило, прочитав главу и потратив некоторое время на изучение исходного кода примеров, читатель сможет создать аналогичный пример самостоятельно. Фактически, попытка самостоятельно создать собственное приложение, используя материал главы и пример кода в качестве справочника, будет хорошим упражнением.

Кроме учебных примеров, сайт содержит доску объявлений и чат. Мы побуждаем читателей связываться друг с другом и размещать вопросы о темах, которые им непонятны или требуют дополнительных объяснений. В большинстве случаев альтернативная точка зрения и объяснение концепции сократят время, необходимое для ее постижения.

И, наконец, мы планируем размещать на сайте дополнительные примеры программ и учебные материалы, касающиеся тех тем, которые по той или иной причине не затрагивались в книге. Также если обратная связь с читателями покажет, что они испытывают трудности в постижении определенной концепции, на сайт могут быть добавлены дополнительные примеры и учебные материалы.

Сопроводительные файлы можно также загрузить с сайта *www.wordware.com/files/dx9*.

Часть I
Математическая подготовка

В этой подготовительной части мы познакомим вас с математическими инструментами, которые будут применяться в остальной части книги. Главными темами обсуждения являются векторы, матрицы и преобразования, которые применяются почти в каждом приведенном в книге примере программы. Кроме того рассматриваются плоскости и лучи, поскольку некоторые программы из книги ссылаются на эти понятия; при первом чтении книги данные разделы можно пропустить.

Обсуждение ведется в легком и неформальном стиле, чтобы материал был доступен читателям с разным уровнем знаний математики. Тем, кто хочет изучить затронутые здесь темы более подробно и глубоко лучше всего обратиться к учебнику линейной алгебры. Те, кто уже изучал линейную алгебру, найдут первую часть книги очень легкой для чтения и могут использовать ее для освежения своих знаний, если такая потребность вдруг возникнет.

Кроме изучения математики, мы рассмотрим используемые для моделирования этих математических объектов классы D3DX и функции, применяемые для выполнения конкретных математических операций.

Цели

- Изучить геометрию и алгебру векторов и их применение в трехмерной компьютерной графике.
- Изучить матрицы, их алгебру и как они используются в трехмерной геометрии для преобразований.
- Изучить, как с помощью алгебры можно моделировать плоскости и лучи, и их применение в трехмерной графике.
- Познакомиться с используемым для выполнения операций трехмерной математики подмножеством классов и функций, предоставляемым библиотекой D3DX.

Векторы в трехмерном пространстве

Геометрическим представлением вектора является направленный отрезок прямой линии, что показано на рис. 1. У каждого вектора есть два свойства: *длина* (также называемая *модулем* или *нормой* вектора) и *направление*. Благодаря этому векторы очень удобны для моделирования физических величин, которые характеризуются модулем и направлением. Например, в главе 14 мы реализуем систему частиц. При этом мы будем использовать векторы для моделирования скорости и ускорения наших частиц. С другой стороны, в трехмерной компьютерной графике векторы часто используются только для моделирования направления. Например, нам часто требуется указать направление распространения световых лучей, ориентацию грани или направление камеры, глядящей на трехмерный мир. Векторы обеспечивают удобный механизм задания направления в трехмерном пространстве.

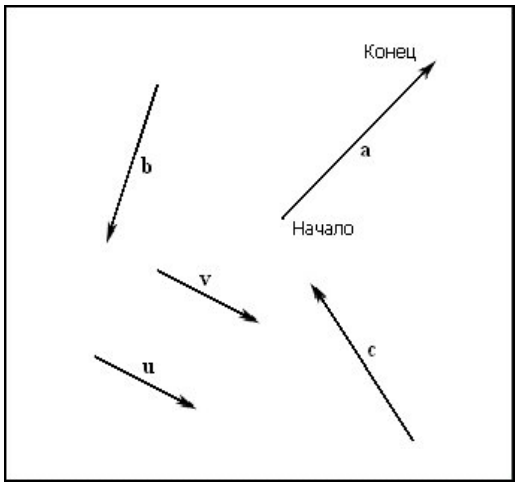

Рис. 1. *Свободные векторы, определенные независимо от системы координат*

Поскольку местоположение не является характеристикой вектора, два вектора с одинаковой длиной и указывающие в одном и том же направлении считаются равными, даже если они расположены в различных местах. Обратите внимание, что два таких вектора будут параллельны друг другу. Например, на рис. 1 векторы **u** и **v** равны.

На рис. 1 видно, что обсуждние векторов может вестись без упоминания системы координат, поскольку всю значимую информацию, — длину и направление, — вектор содержит в себе. Добавление системы координат не добавляет информации в вектор; скорее можно говорить, что вектор, значения которого являются его неотъемлимой частью, просто описан относительно конкретной системы координат. И если мы изменим систему координат, мы только опишем *тот же самый* вектор относительно другой системы.

Отметив этот важный момент, мы перейдем к изучению того, как векторы описываются в левосторонней трехмерной декартовой системе координат. На рис. 2 показаны левосторонняя и правосторонняя системы координат. Различие

между ними — положительное направление оси Z. В левосторонней системе координат положительное направление оси Z погружается в страницу. В правосторонней системе координат положительное направление оси Z направлено от страницы.

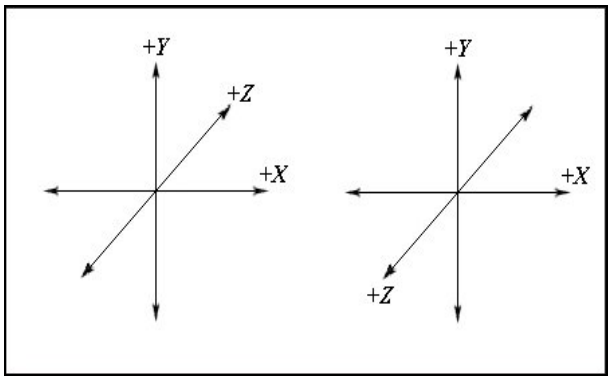

Рис. 2. *Слева изображена левосторонняя система координат. Обратите внимание, что положительное направление оси Z направлено вглубь страницы. Справа изображена правостороняя система координат. Здесь положительное направление оси Z направлено от страницы*

Поскольку местоположение вектора не изменяет его свойств, мы можем перенести векторы таким образом, чтобы начало каждого из них совпадало с началом координат выбранной координатной системы. Когда начало вектора совпадает с началом координат, говорят, что вектор находится в *стандартной позиции*. Таким образом, если вектор находится в стандартной позиции, мы можем описать его, указав только координаты конечной точки. Мы будем называть эти координаты *компонентами* вектора. На рис. 3 показаны векторы, изображенные на рис. 1, которые были перемещены в стандартные позиции.

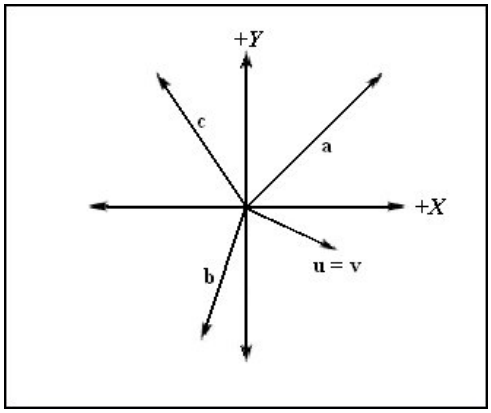

Рис. 3. *Векторы в стандартной позиции, определенные в указанной системе координат. Обратите внимание, что векторы* ***u*** *и* ***v*** *полностью совпадают друг с другом потому что они равны*

ПРИМЕЧАНИЕ Поскольку мы описываем находящийся в стандартной позиции вектор, указывая его конечную точку, как если бы мы описывали отдельную точку, легко перепутать точку и вектор. Чтобы подчеркнуть различия между этими двумя понятиями, мы вновь приведем определение каждого из них. Точка описывает только местоположение в системе координат, в то время как вектор описывает величину и направление.

Мы будем пользоваться для обозначения векторов полужирными строчными буквами, но иногда будем применять и полужирные заглавные буквы. Вот пример двух-, трех- и четырехмерных векторов соответственно: **u** = (u_x, u_y), **N** = (N_x, N_y, N_z), **c** = (c_x, c_y, c_z, c_w).

Теперь мы введем четыре специальных трехмерных вектора, которые показаны на рис. 4. Первый из них называется *нулевым вектором*, и значения всех его компонент равны нулю; мы будем обозначать такой вектор выделенным полужирным шрифтом нулем: **0** = (0, 0, 0). Следующие три специальных вектора называются единичными базовыми векторами (базовыми ортами) трехмерной системы координат. Эти векторы, направленные вдоль осей X, Y и Z нашей координатной системы, мы будем называть **i**, **j** и **k** соответственно. Модуль этих векторов равен единице, а определение выглядит следующим образом: **i** = (1, 0, 0), **j** = (0, 1, 0), **k** = (0, 0, 1).

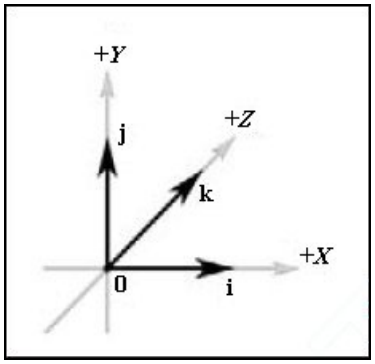

Рис. 4. *Нулевой вектор и базовые орты трехмерной системы координат*

ПРИМЕЧАНИЕ Вектор, длина которого равна единице, называется *единичным вектором* или *ортом*.

В библиотеке D3DX для представления векторов в трехмерном пространстве мы можем воспользоваться классом **D3DXVECTOR3**. Его определение выглядит следующим образом:

```
typedef struct D3DXVECTOR3 : public D3DVECTOR {
public:
    D3DXVECTOR3() {};
    D3DXVECTOR3( CONST FLOAT * );
    D3DXVECTOR3( CONST D3DVECTOR& );
    D3DXVECTOR3( FLOAT x, FLOAT y, FLOAT z );

    // приведение типа
    operator FLOAT* ();
    operator CONST FLOAT* () const;

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

```
    // унарные операторы
    D3DXVECTOR3 operator + () const;
    D3DXVECTOR3 operator - () const;

    // бинарные операторы
    D3DXVECTOR3 operator + ( CONST D3DXVECTOR3& ) const;
    D3DXVECTOR3 operator - ( CONST D3DXVECTOR3& ) const;
    D3DXVECTOR3 operator * ( FLOAT ) const;
    D3DXVECTOR3 operator / ( FLOAT ) const;
    friend D3DXVECTOR3 operator * ( FLOAT,
                                    CONST struct D3DXVECTOR3& );
    BOOL operator == ( CONST D3DXVECTOR3& ) const;
    BOOL operator != ( CONST D3DXVECTOR3& ) const;
} D3DXVECTOR3, *LPD3DXVECTOR3;
```

Обратите внимание, что **D3DXVECTOR3** наследует компоненты от **D3DVECTOR**, определение которого выглядит следующим образом:

```
typedef struct _D3DVECTOR {
    float x;
    float y;
    float z;
} D3DVECTOR;
```

Так же, как и у скалярных величин, у векторов есть собственная арифметика, что видно из наличия описаний математических операций в определении класса **D3DXVECTOR3**. Возможно, сейчас вы не знаете, что делают эти методы. В следующих подразделах мы рассмотрим эти операции с векторами, другие вспомогательные функции работы с векторами из библиотеки D3DX и некоторые важные особенности обработки векторов.

ПРИМЕЧАНИЕ Хотя основной интерес для нас представляют векторы в трехмерном пространстве, занимаясь программированием трехмерной графики мы будем иногда сталкиваться с векторами в двухмерном и четырехмерном пространствах. Библиотека D3DX предоставляет классы **D3DXVECTOR2** и **D3DXVECTOR4**, предназначенные для представления векторов в двухмерном и четырехмерном пространствах соответственно. Векторы в пространствах с другим количеством измерений обладают теми же свойствами, что и векторы в трехмерном пространстве, а именно — длиной и направлением, отличается только количество измерений. Кроме того, математические операции с векторами, за исключением векторного произведения (см. раздел «Векторное произведение», далее в этой главе), которое определено только для трехмерной системы координат, могут быть обобщены для векторов любой размерности. Таким образом, за исключением векторного произведения, все операции, которые мы обсуждаем для векторов в трехмерном пространстве, распространяются и на векторы в двухмерном, четырехмерном и даже n-мерном пространствах.

Равенство векторов

В геометрии два вектора считаются равными, если они указывают в одном и том же направлении и имеют одинаковую длину. В алгебре говорят, что векторы равны, если у них одинаковое количество измерений и их соответствующие компоненты равны. Например, $(u_x, u_y, u_z) = (v_x, v_y, v_z)$ если $u_x = v_x$, $u_y = v_y$ и $u_z = v_z$.

В коде мы можем проверить равны ли два вектора, используя перегруженный оператор равенства:

```
D3DXVECTOR u(1.0f, 0.0f, 1.0f);
D3DXVECTOR v(0.0f, 1.0f, 0.0f);
if( u == v ) return true;
```

Аналогичным образом, можно убедиться, что два вектора не равны, используя перегруженный оператор неравенства:

```
if( u != v ) return true;
```

ПРИМЕЧАНИЕ Сравнивая числа с плавающей точкой следует быть очень аккуратным, поскольку из-за погрешностей округления, два числа с плавающей точкой, которые должны быть равными, могут слегка отличаться. По этой причине мы проверяем приблизительное равенство чисел с плавающей точкой. Для этого мы определили константу **EPSILON**, содержащую очень маленькое значение, которое будет служить «буфером». Мы будем говорить, что два числа приблизительно равны, если разница между ними меньше **EPSILON**. Другими словами, **EPSILON** дает нам некий допуск для ошибок округления чисел с плавающей точкой. Приведенная ниже функция показывает, как **EPSILON** может использоваться при проверке равенства двух чисел с плавающей точкой:

```
const float EPSILON = 0.001f;
bool Equals(float lhs, float rhs)
{
   // если lhs == rhs разность должна быть равна нулю
   return fabs(lhs - rhs) < EPSILON ? true : false;
}
```

Об этом не надо беспокоиться, работая с классом **D3DXVECTOR**, поскольку перегруженные операции сравнения все сделают за нас, но очень важно знать об этой особенности сравнения чисел с плавающей точкой.

Вычисление модуля вектора

В геометрии модулем вектора называется длина направленного отрезка линии. В алгебре, зная компоненты вектора мы можем вычислить его модуль по следующей формуле:

$$(1)\ \ |\mathbf{u}| = \sqrt{u_x^2 + u_y^2 + u_z^2}$$

Вертикальные линии в **|u|** обозначают модуль **u**.

ПРИМЕР Вычислите модуль векторов **u** = (1, 2, 3) и **v** = (1, 1).

Решение:

Для вектора **u** мы получаем:

$$(2)\ |\mathbf{u}| = \sqrt{\mathbf{1}^2 + \mathbf{2}^2 + \mathbf{3}^2} = \sqrt{\mathbf{1} + \mathbf{4} + \mathbf{9}} = \sqrt{\mathbf{14}}$$

Обобщив формулу (1) для двухмерного пространства, для вектора **v** мы получим:

$$(3)\ |\mathbf{v}| = \sqrt{\mathbf{1}^2 + \mathbf{1}^2} = \sqrt{\mathbf{2}}$$

Работая с библиотекой D3DX, для вычисления модуля вектора мы можем применять следующую функцию:

```
FLOAT D3DXVec3Length(     // Возвращает модуль
    CONST D3DXVECTOR3* pV // Вектор, чью длину мы вычисляем
);

D3DXVECTOR3 v(1.0f, 2.0f, 3.0f);
float magnitude = D3DXVec3Length(&v); // = sqrt(14)
```

Нормализация вектора

В результате нормализации получается вектор, направление которого совпадает с исходным, а модуль равен единице (единичный вектор). Чтобы нормализовать произвольный вектор, достаточно разделить каждый компонент вектора на модуль вектора, как показано ниже:

$$\hat{\mathbf{u}} = \frac{\mathbf{u}}{|\mathbf{u}|} = \left(\frac{u_x}{|\mathbf{u}|}, \frac{u_y}{|\mathbf{u}|}, \frac{u_z}{|\mathbf{u}|} \right)$$

Мы отмечаем единичный вектор, помещая над его обозначением символ **^**: $\hat{\mathbf{u}}$.

ПРИМЕР Нормализуйте векторы **u** = (1, 2, 3) и **v** = (1, 1).

Решение:

Из приведенных выше формул (2) и (3) мы знаем, что **|u|** = $\sqrt{14}$ и **|v|** = $\sqrt{2}$, поэтому:

$$\hat{\mathbf{u}} = \frac{\mathbf{u}}{\sqrt{\mathbf{14}}} = \left(\frac{\mathbf{1}}{\sqrt{\mathbf{14}}}, \frac{\mathbf{2}}{\sqrt{\mathbf{14}}}, \frac{\mathbf{3}}{\sqrt{\mathbf{14}}} \right)$$

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{\sqrt{\mathbf{2}}} = \left(\frac{\mathbf{1}}{\sqrt{\mathbf{2}}}, \frac{\mathbf{1}}{\sqrt{\mathbf{2}}} \right)$$

В библиотеке D3DX для нормализации векторов применяется следующая функция:

```
D3DXVECTOR3 *D3DXVec3Normalize(
    D3DXVECTOR3* pOut,    // Результат
    CONST D3DXVECTOR3* pV // Нормализуемый вектор
);
```

ПРИМЕЧАНИЕ Эта функция возвращает указатель на результат, который может быть передан в качестве параметра другой функции. В большинстве случаев, за исключением явно указанных, математические функции библиотеки D3DX возвращают указатель на результат. Мы не будем явно говорить это для каждой функции.

Сложение векторов

Мы можем сложить два вектора, сложив их соответствующие компоненты; обратите внимание, что размерность складываемых векторов должна быть одинаковой:

$$\mathbf{u} + \mathbf{v} = \left(u_x + v_x, u_y + v_y, u_z + v_z\right)$$

Геометрическая интерпретация сложения векторов показана на рис. 5.

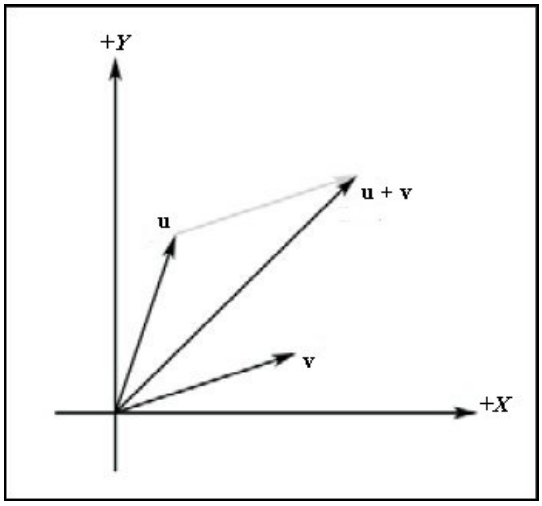

Рис. 5. *Сложение векторов. Обратите внимание, как мы выполняем параллельный перенос вектора* ***v*** *таким образом, чтобы его начало совпало с концом вектора* ***u****; суммой будет вектор начало которого совпадает с началом вектора* ***u****, а конец совпадает с концом перенесенного вектора* ***v***

В коде для сложения двух векторов мы будем применять перегруженый оператор сложения:

```
D3DXVECTOR3 u(2.0f, 0.0f, 1.0f);
D3DXVECTOR3 v(0.0f, -1.0f, 5.0f);

// (2.0 + 0.0, 0.0 + (-1.0), 1.0 + 5.0)
D3DXVECTOR3 sum = u + v; // = (2.0f, -1.0f, 6.0f)
```

Вычитание векторов

Аналогично сложению, вычитание векторов осуществляется путем вычитания их отдельных компонент. Опять же оба вектора должны иметь одинаковую размерность.

$$\mathbf{u} - \mathbf{v} = \mathbf{u} + (-\mathbf{v}) = (u_x - v_x, u_y - v_y, u_z - v_z)$$

Геометрическая интерпретация вычитания векторов показана на рис. 6.

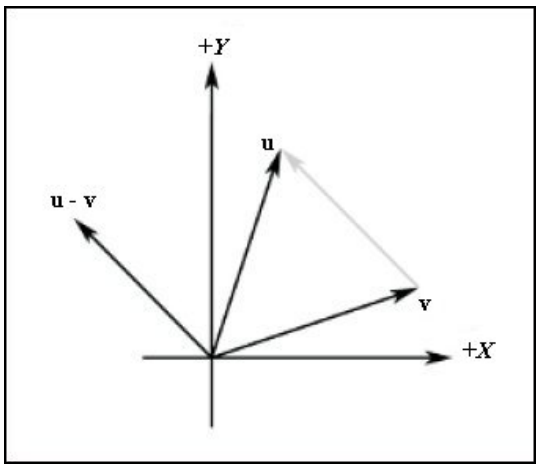

Рис. 6. *Вычитание векторов*

В коде для вычитания двух векторов мы будем применять перегруженый оператор вычитания:

```
D3DXVECTOR3 u(2.0f, 0.0f, 1.0f);
D3DXVECTOR3 v(0.0f, -1.0f, 5.0f);

D3DXVECTOR3 difference = u - v; // = (2.0f, 1.0f, -4.0f)
```

Как видно на рис. 6, операция вычитания векторов возвращает вектор, начало которого совпадает с концом вектора **v**, а конец — с концом вектора **u**. Если мы интерпретируем компоненты **u** и **v** как координаты точек, то результатом вычитания будет вектор, направленный от одной точки к другой. Это очень удобная операция, поскольку нам часто будет необходимо найти вектор, описывающий направление от одной точки к другой.

Умножение вектора на скаляр

Как видно из названия раздела, мы можем умножать вектор на скаляр, в результате чего происходит масштабирование вектора. Если масштабный множитель положителен, направление вектора не меняется. Если же множитель отрицателен, то направление вектора изменяется на противоположное (инвертируется).

$$k\mathbf{u} = (ku_x, ku_y, ku_z)$$

Класс **D3DXVECTOR3** предоставляет оператор умножения вектора на скаляр:

```
D3DXVECTOR3 u(1.0f, 1.0f, -1.0f);
D3DXVECTOR3 scaledVec = u * 10.0f; // = (10.0f, 10.0f, -10.0f)
```

Скалярное произведение векторов

Скалярное произведение векторов — это первая из двух определенных в векторной алгебре операций умножения. Вычисляется такое произведение следующим образом:

$$\mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y + u_z v_z = s$$

У приведенной выше формулы нет очевидной геометрической интерпретации. Используя теорему косинусов[1], мы получим отношение $\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}||\mathbf{v}| \cos \varphi$, говорящее, что скалярное произведение двух векторов равно произведению косинуса угла между векторами на модули векторов. Следовательно, если **u** и **v** — единичные векторы, их скалярное произведение равно косинусу угла между ними.

Вот некоторые полезные свойства скалярного произведения:

- Если $\mathbf{u} \cdot \mathbf{v} = 0$, значит $\mathbf{u} \perp \mathbf{v}$.
- Если $\mathbf{u} \cdot \mathbf{v} > 0$, значит угол φ между двумя векторами меньше 90 градусов.
- Если $\mathbf{u} \cdot \mathbf{v} < 0$, значит угол φ между двумя векторами больше 90 градусов.

ПРИМЕЧАНИЕ Символ $\perp$ обозначает «ортогональный» или (что то же самое) «перпендикулярный».

Для вычисления скалярного произведения двух векторов в библиотеке D3DX предназначена следующая функция:

```
FLOAT D3DXVec3Dot( // Возвращает результат
    CONST D3DXVECTOR3* pV1, // Левый операнд
    CONST D3DXVECTOR3* pV2  // Правый операнд
);

D3DXVECTOR3 u(1.0f, -1.0f, 0.0f);
D3DXVECTOR3 v(3.0f, 2.0f, 1.0f);

// 1.0 * 3.0 + -1.0 * 2.0 + 0.0 * 1.0
// = 3.0 + -2.0

float dot = D3DXVec3Dot(&u, &v); // = 1.0
```

[1] Теорема косинусов определяет зависимость между сторонами и углами треугольника. Она утверждает, что во всяком треугольнике квадрат длины стороны равен сумме квадратов двух других сторон без удвоенного произведения длин этих сторон на косинус угла между ними. Если угол прямой, то теорема косинусов переходит в теорему Пифагора, т.к. косинус прямого угла равен 0.

Векторное произведение

Второй формой операции умножения, определенной в векторной алгебре, является векторное произведение. В отличие от скалярного произведения, результатом которого является число, результатом векторного произведения будет вектор. Векторным произведением двух векторов **u** и **v** будет вектор **p**, являющийся взаимно перпендикулярным для векторов **u** и **v**. Это означает, что вектор **p** перпендикулярен вектору **u** и одновременно вектор **p** перпендикулярен вектору **v**.

Вычисляется векторное произведение по следующей формуле:

$$\mathbf{p} = \mathbf{u} \times \mathbf{v} = \left[\left(u_y v_z - u_z v_y\right), \left(u_z v_x - u_x v_z\right), \left(u_x v_y - u_y v_x\right)\right]$$

В компонентной форме вычисление выглядит так:

$$p_x = \left(u_y v_z - u_z v_y\right)$$
$$p_y = \left(u_z v_x - u_x v_z\right)$$
$$p_z = \left(u_x v_y - u_y v_x\right)$$

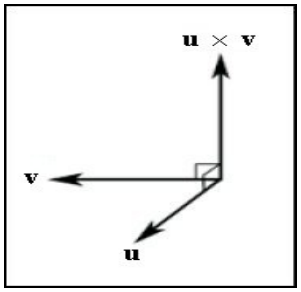

Рис. 7. *Векторное произведение. Вектор* ***p*** *=* ***u*** *×* ***v*** *перпендикулярен как вектору* ***u****, так и вектору* ***v***

ПРИМЕР

Вычислите **j** = **k** × **i** = (0, 0, 1) × (1, 0, 0) и проверьте, что вектор **j** перпендикулярен как вектору **i**, так и вектору **k**.

Решение:

$$j_x = \left(\mathbf{0} * \mathbf{0} - \mathbf{1} * \mathbf{0}\right) = \mathbf{0}$$
$$j_y = \left(\mathbf{1} * \mathbf{1} - \mathbf{0} * \mathbf{0}\right) = \mathbf{1}$$
$$j_z = \left(\mathbf{0} * \mathbf{0} - \mathbf{0} * \mathbf{1}\right) = \mathbf{0}$$

Таким образом, **j** = (0, 1, 0). Вспомните, в предыдущем разделе «Скалярное произведение векторов» говорилось, что если **u** · **v** = 0, значит **u** ⊥ **v**. Поскольку **j** · **k** = 0 и **j** · **i** = 0, мы знаем что вектор **j** перпендикулярен как вектору **k**, так и вектору **i**.

Для вычисления векторного произведения двух векторов в библиотеке D3DX предназначена следующая функция:

```
D3DXVECTOR3 *D3DXVec3Cross(
    D3DXVECTOR3* pOut,      // Результат
    CONST D3DXVECTOR3* pV1, // Левый операнд
    CONST D3DXVECTOR3* pV2  // Правый операнд
);
```

Как явствует из рис. 7, вектор **–p** также взаимно перпендикулярен векторам **u** и **v**. Какой из векторов, **p** или **–p** будет возвращен в качестве результата векторного произведения определяется порядком операндов. Другими словами, $\mathbf{u} \times \mathbf{v} = -(\mathbf{v} \times \mathbf{u})$. Это значит, что операция векторного произведения не является коммутативной. Определить, какой вектор будет возвращен в качестве результата, можно с помощью *правила левой руки*. (Мы используем правило левой руки, поскольку работаем с левосторонней системой координат. Если бы у нас была правосторонняя система координат, пришлось бы воспользоваться правилом правой руки.) Если расположить пальцы левой руки вдоль первого вектора, а ладонь руки — вдоль второго, отогнутый на 90 градусов большой палец укажет направление результирующего вектора.

Матрицы

В этом разделе мы сосредоточимся на математике матриц. Их использование в трехмерной компьютерной графике будет рассмотрено в следующем разделе.

Матрицей $m \times n$ называется прямоугольный массив чисел, состоящий из m строк и n столбцов. Количество строк и столбцов определяет размер матрицы. Отдельный элемент матрицы идентифицируется путем указания его строки и столбца в состоящем из двух элементов спсике индексов; первый индекс определяет строку, а второй — столбец. Ниже в качестве примера приведены матрицы **M** размером 3 × 3, **B** размером 2 × 4 и **C** размером 3 × 2:

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \end{bmatrix} \quad \mathbf{C} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{bmatrix}$$

В большинстве случаев для обозначения матриц мы будем использовать заглавные полужирные буквы. Иногда матрицы состоят из единственной строки или единственного столбца. Чтобы отличать такие матрицы, мы дадим им специальные имена: *вектор-строка* (*row vector*) и *вектор-столбец* (*column vector*). Вот примеры таких векторов:

$$\mathbf{v} = \begin{bmatrix} v_1, & v_2, & v_3, & v_4 \end{bmatrix} \quad \mathbf{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix}$$

Для элементов вектора-строки и вектора-столбца необходим только один индекс. Иногда для идентификации элемента строки или столбца в качестве индекса мы будем использовать буквы.

Равенство, умножение матрицы на скаляр и сложение матриц

Для пояснения рассматриваемых терминов в данном разделе будут использованые следующие четыре матрицы:

$$\mathbf{A}=\begin{bmatrix}1 & 5\\ -2 & 3\end{bmatrix}\quad \mathbf{B}=\begin{bmatrix}6 & 2\\ 5 & -8\end{bmatrix}\quad \mathbf{C}=\begin{bmatrix}1 & 5\\ -2 & 3\end{bmatrix}\quad \mathbf{D}=\begin{bmatrix}1 & 2 & -1 & 3\\ -6 & 3 & 0 & 0\end{bmatrix}$$

- Две матрицы считаются равными, если они имеют одинаковую размерность и их соответствующие элементы равны. Например, $\mathbf{A}=\mathbf{C}$, поскольку матрицы $\mathbf{A}$ и $\mathbf{C}$ имеют одинаковую размерность и их соответствующие элементы равны. Мы говорим, что $\mathbf{A}\neq\mathbf{B}$ и $\mathbf{A}\neq\mathbf{D}$ поскольку у этих матриц либо разная размерность, либо не равны соответствующие элементы.
- Мы можем умножить матрицу на скаляр для чего нам необходимо умножить каждый элемент матрицы на данный скаляр. Например, умножив $\mathbf{D}$ на скаляр k получим:

$$k\mathbf{D}=\begin{bmatrix}k*1 & k*2 & k*-1 & k*3\\ k*-6 & k*3 & k*0 & k*0\end{bmatrix}$$

  Если $k = 2$, получим:

$$k\mathbf{D}=2\mathbf{D}=\begin{bmatrix}2*1 & 2*2 & 2*-1 & 2*3\\ 2*-6 & 2*3 & 2*0 & 2*0\end{bmatrix}=\begin{bmatrix}2 & 4 & -2 & 6\\ -12 & 6 & 0 & 0\end{bmatrix}$$

- Можно сложить две матрицы, но только в том случае, если у них одинаковая размерность. Сумма вычисляется путем сложения соответствующих элементов матриц. Например:

$$\mathbf{A}+\mathbf{B}=\begin{bmatrix}1 & 5\\ -2 & 3\end{bmatrix}+\begin{bmatrix}6 & 2\\ 5 & -8\end{bmatrix}=\begin{bmatrix}1+6 & 5+2\\ -2+5 & 3+(-8)\end{bmatrix}=\begin{bmatrix}7 & 7\\ 3 & -5\end{bmatrix}$$

- Аналогично сложению можно выполнять вычитание двух матриц, имеющих одинаковую размерность. Вычитание матриц иллюстрирует следующий пример:

$$\mathbf{A}-\mathbf{B}=\mathbf{A}+(-\mathbf{B})=\begin{bmatrix}1 & 5\\ -2 & 3\end{bmatrix}-\begin{bmatrix}6 & 2\\ 5 & -8\end{bmatrix}=\begin{bmatrix}1 & 5\\ -2 & 3\end{bmatrix}+\begin{bmatrix}-6 & -2\\ -5 & 8\end{bmatrix}=\begin{bmatrix}1-6 & 5-2\\ -2-5 & 3+8\end{bmatrix}=\begin{bmatrix}-5 & 3\\ -7 & 11\end{bmatrix}$$

Умножение

Умножение матриц это наиболее важная операция, которая постоянно используется в трехмерной компьютерной графике. Именно умножение матриц позволяет осуществлять преобразование векторов и комбинировать несколько преобразований в одно. Преобразования будут рассмотрены в следующем разделе.

Чтобы получить произведение матриц **AB** необходимо чтобы количество столбцов матрицы **A** было равно количеству строк матрицы **B**. Если условие выполняется, произведение матриц определено. Рассмотрим представленные ниже матрицы **A** и **B**, с размерностью 2 × 3 и 3 × 3 соответственно:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

Как видите, произведение **AB** определено поскольку количество столбцов матрицы **A** равно количеству строк матрицы **B**. Обратите внимание, что произведение **BA**, получаемое в результате перестановки множителей, не определено, потому что количество столбцов матрицы **B** *не равно* количеству строк матрицы **A**. Это говорит о том, что в обшем случае операция умножения матриц не коммутативна (то есть $\mathbf{AB} \neq \mathbf{BA}$). Мы говорим «в общем случае не коммутативна» по той причине, что существует ряд частных случаев в которых операция умножения матриц ведет себя как коммутативная.

После того, как мы узнали в каких случаях произведение матриц определено, можно дать определение операции умножения матриц: если **A** — это матрица $m \times n$, а **B** — матрица $n \times p$, то их произведением будет матрица **C**, размером $m \times p$, в которой элемент $\mathbf{c}_{ij}$ находится как скалярное произведение i-го вектора-строки матрицы **A** и j-го вектора-столбца матрицы **B**:

$$(4) \quad c_{ij} = \mathbf{a}_i \cdot \mathbf{b}_j$$

В этой формуле $\mathbf{a}_i$ обозначает i-ый вектор-строку в матрице **A**, а $\mathbf{b}_j$ — j-ый вектор-столбец матрицы **B**.

Давайте для примера вычислим произведение:

$$\mathbf{AB} = \begin{bmatrix} \mathbf{4} & \mathbf{1} \\ \mathbf{-2} & \mathbf{1} \end{bmatrix} \begin{bmatrix} \mathbf{1} & \mathbf{3} \\ \mathbf{2} & \mathbf{1} \end{bmatrix}$$

Произведение определено, поскольку количество столбцов матрицы **A** равно количеству строк матрицы **B**. Кроме того, обратите внимание, что размер полученной в результате матрицы — 2 × 2. Согласно формуле (4) получаем:

$$\mathbf{AB}=\begin{bmatrix}\mathbf{4} & \mathbf{1}\\ \mathbf{-2} & \mathbf{1}\end{bmatrix}\begin{bmatrix}\mathbf{1} & \mathbf{3}\\ \mathbf{2} & \mathbf{1}\end{bmatrix}=\begin{bmatrix}\mathbf{a}_1\cdot\mathbf{b}_1 & \mathbf{a}_1\cdot\mathbf{b}_2\\ \mathbf{a}_2\cdot\mathbf{b}_1 & \mathbf{a}_2\cdot\mathbf{b}_2\end{bmatrix}=\begin{bmatrix}(\mathbf{4}\ \ \mathbf{1})\cdot(\mathbf{1}\ \ \mathbf{2}) & (\mathbf{4}\ \ \mathbf{1})\cdot(\mathbf{3}\ \ \mathbf{1})\\ (\mathbf{-2}\ \ \mathbf{1})\cdot(\mathbf{1}\ \ \mathbf{2}) & (\mathbf{-2}\ \ \mathbf{1})\cdot(\mathbf{3}\ \ \mathbf{1})\end{bmatrix}=\begin{bmatrix}\mathbf{6} & \mathbf{13}\\ \mathbf{0} & \mathbf{-5}\end{bmatrix}$$

В качестве упражнения проверьте, что в данном случае AB ≠ BA.

И еще один, более общий, пример:

$$\mathbf{AB}=\begin{bmatrix}a_{11} & a_{12} & a_{13}\\ a_{21} & a_{22} & a_{23}\end{bmatrix}\begin{bmatrix}b_{11} & b_{12}\\ b_{21} & b_{22}\\ b_{31} & b_{32}\end{bmatrix}=\begin{bmatrix}a_{11}b_{11}+a_{12}b_{21}+a_{13}b_{31} & a_{11}b_{12}+a_{12}b_{22}+a_{13}b_{32}\\ a_{21}b_{11}+a_{22}b_{21}+a_{23}b_{31} & a_{21}b_{12}+a_{22}b_{22}+a_{23}b_{32}\end{bmatrix}=\mathbf{C}$$

Единичная матрица

Существует особая матрица, называемая *единичной матрицей* (*identity matrix*). Это квадратная матрица все элементы которой равны нулю, за исключением тех, что расположены на главной диагонали — эти элементы равны единице. Ниже приведены примеры единичных матриц размером 2 × 2, 3 × 3 и 4 × 4:

$$\begin{bmatrix}\mathbf{1} & \mathbf{0}\\ \mathbf{0} & \mathbf{1}\end{bmatrix}\quad\begin{bmatrix}\mathbf{1} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{1} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{1}\end{bmatrix}\quad\begin{bmatrix}\mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1}\end{bmatrix}$$

Единичная матрица действует как мультипликативное тождество:

$$\mathbf{MI}=\mathbf{IM}=\mathbf{M}$$

Следовательно, операция умножения на единичную матрицу не изменяет исходную матрицу. Более того, умножение на единичную матрицу это один из случаев, когда операция умножения матриц является коммутативной. Вы можете думать о единичной матрице как о числе 1 для матриц.

ПРИМЕР

Проверьте, что в результате умножения матрицы $\mathbf{M}=\begin{bmatrix}1 & 2\\ 0 & 4\end{bmatrix}$ на единичную матрицу размером 2 × 2 получается матрица **M**.

Решение:

$$\begin{bmatrix}\mathbf{1} & \mathbf{2}\\ \mathbf{0} & \mathbf{4}\end{bmatrix}\begin{bmatrix}\mathbf{1} & \mathbf{0}\\ \mathbf{0} & \mathbf{1}\end{bmatrix}=\begin{bmatrix}(\mathbf{1}\ \ \mathbf{2})\cdot(\mathbf{1}\ \ \mathbf{0}) & (\mathbf{1}\ \ \mathbf{2})\cdot(\mathbf{0}\ \ \mathbf{1})\\ (\mathbf{0}\ \ \mathbf{4})\cdot(\mathbf{1}\ \ \mathbf{0}) & (\mathbf{0}\ \ \mathbf{4})\cdot(\mathbf{0}\ \ \mathbf{1})\end{bmatrix}=\begin{bmatrix}\mathbf{1} & \mathbf{2}\\ \mathbf{0} & \mathbf{4}\end{bmatrix}$$

Инвертирование матриц

В математике матриц нет аналога операции деления, но зато есть мультипликативная операция инвертирования. Приведенный ниже список обобщает важные особенности инвертирования:

- Инвертировать можно только квадратные матрицы, так что когда мы говорим об инвертировании матрицы, подразумевается, что мы имеем дело с квадратной матрицей.
- В результате инвертирования матрицы $\mathbf{M}$ размером $n \times n$ получается матрица размером $n \times n$, которую мы будем обозначать $\mathbf{M}^{-1}$.
- Не всякую квадратную матрицу можно инвертировать.
- Если перемножить исходную и инвертированную матрицы, получится единичная матрица: $\mathbf{M}\mathbf{M}^{-1} = \mathbf{M}^{-1}\mathbf{M} = \mathbf{I}$. Обратите внимание, что в случае перемножения исходной и инвертированной матриц операция умножения матриц коммутативна.

Инверсия матриц применяется для нахождения искомой матрицы в уравнениях. Для примера возьмем выражение $\mathbf{p}' = \mathbf{p}\mathbf{R}$ и предположим, что нам известны $\mathbf{p}'$ и $\mathbf{R}$, а требуется найти $\mathbf{p}$. Сначала вычислим $\mathbf{R}^{-1}$ (подразумевается, что эта матрица существует). Получив $\mathbf{R}^{-1}$ можно вычислить $\mathbf{p}$ по следующему алгоритму:

$$\mathbf{p}'\mathbf{R}^{-1} = \mathbf{p}\left(\mathbf{R}\mathbf{R}^{-1}\right)$$
$$\mathbf{p}'\mathbf{R}^{-1} = \mathbf{p}\mathbf{I}$$
$$\mathbf{p}'\mathbf{R}^{-1} = \mathbf{p}$$

Описание способа вычисления инвертированной матрицы выходит за рамки этой книги, но вы можете его найти в любом учебнике линейной алгебры. В разделе «Базовые преобразования» мы получим инверсию нескольких матриц, с которыми будем работать. В раздеде «Матрицы в библиотеке D3DX» мы познакомимся с функцией библиотеки D3DX, которая может инвертировать матрицу за нас.

Завершая раздел об инвертировании матриц представим вам одно полезное свойство, касающееся инвертирования произведения: $(\mathbf{A}\mathbf{B})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$. Здесь подразумеваеся, что матрицы $\mathbf{A}$ и $\mathbf{B}$ могут быть инвертированы и что обе они — квадратные матрицы одинакового размера.

Транспонирование матриц

Транспонирование матрицы осуществляется путем перестановки ее строк и столбцов. Следовательно, результатом транспонирования матрицы $m \times n$ будет матрица $n \times m$. Результат транспонирования матрицы $\mathbf{M}$ мы будем обозначать $\mathbf{M}^T$.

ПРИМЕР Транспонируйте следующие две матрицы:

$$\mathbf{A} = \begin{bmatrix} \mathbf{2} & \mathbf{-1} & \mathbf{8} \\ \mathbf{3} & \mathbf{6} & \mathbf{-4} \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

Решение:

Напомним, что транспонирование матрицы осуществляется путем перестановки ее строк и столбцов. Следовательно:

$$\mathbf{A}^T = \begin{bmatrix} \mathbf{2} & \mathbf{3} \\ \mathbf{-1} & \mathbf{6} \\ \mathbf{8} & \mathbf{-4} \end{bmatrix} \quad \mathbf{B}^T = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

Матрицы в библиотеке D3DX

Программируя приложения Direct3D мы чаще всего будем использовать матрицы 4 × 4 и векторы-строки 1 × 4. Обратите внимание, что использование матриц двух указанных размеров подразумевает, что определены результаты следующих операций умножения матриц:

- **Умножение вектора-строки на матрицу.** То есть, если **v** — это вектор-строка 1 × 4, а **T** — это матрица 4 × 4, произведение **vT** определено и представляет собой вектор-строку 1 × 4.
- **Умножение матрицы на матрицу.** То есть, если **T** — это матрица 4 × 4 и **R** — это матрица 4 × 4, произведения **TR** и **RT** определены и оба являются матрицами 4 × 4. Обратите внимание, что произведение **TR** не обязательно равно **RT**, поскольку операция умножения матриц не коммутативна.

Для представления вектора-строки 1 × 4 в библиотеке D3DX, мы будем использовать классы векторов **D3DXVECTOR3** и **D3DXVECTOR4**. Конечно, в классе **D3DXVECTOR3** только три компоненты, а не четыре. Однако обычно подразумевается что четвертая компонента равна нулю или единице (более подробно это будет обсуждаться в следующем разделе).

Для представления матриц 4 × 4 в библиотеке D3DX, мы используем класс **D3DXMATRIX**, определение которого выглядит следующим образом:

```
typedef struct D3DXMATRIX : public D3DMATRIX
{
public:
     D3DXMATRIX() {};
     D3DXMATRIX(CONST FLOAT*);
     D3DXMATRIX(CONST D3DMATRIX&);
```

```
    D3DXMATRIX(FLOAT _11, FLOAT _12, FLOAT _13, FLOAT _14,
               FLOAT _21, FLOAT _22, FLOAT _23, FLOAT _24,
               FLOAT _31, FLOAT _32, FLOAT _33, FLOAT _34,
               FLOAT _41, FLOAT _42, FLOAT _43, FLOAT _44);

    // получение элемента
    FLOAT& operator () (UINT Row, UINT Col);
    FLOAT operator () (UINT Row, UINT Col) const;

    // приведение типа
    operator FLOAT* ();
    operator CONST FLOAT* () const;

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

    // унарные операторы
    D3DXMATRIX operator + () const;
    D3DXMATRIX operator - () const;

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

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

    BOOL operator == (CONST D3DXMATRIX&) const;
    BOOL operator != (CONST D3DXMATRIX&) const;
} D3DXMATRIX, *LPD3DXMATRIX;
```

Класс **D3DXMATRIX** наследует элементы данных от простой структуры **D3DMATRIX**, определенной следующим образом:

```
typedef struct _D3DMATRIX {
    union {
        struct {
            float _11, _12, _13, _14;
            float _21, _22, _23, _24;
            float _31, _32, _33, _34;
            float _41, _42, _43, _44;
        };
        float m[4][4];
    };
} D3DMATRIX;
```

Обратите внимание, что в классе **D3DXMATRIX** есть десятки полезных операторов для проверки равенства, сложения и вычитания матриц, умножения

матрицы на скаляр, преобразования типов и — самое главное — перемножения двух объектов типа **D3DXMATRIX**. Поскольку умножение матриц так важно, приведем пример кода, использующего этот оператор:

```
D3DXMATRIX A(E);       // инициализация A
D3DXMATRIX B(E);       // инициализация B
D3DXMATRIX C = A * B; // C = AB
```

Другим важным оператором класса **D3DXMATRIX** являются скобки, позволяющие легко получить доступ к отдельным элементам матрицы. Обратите внимание, что при использовании скобок нумерация элементов матрицы начинается с нуля, подобно нумерации элементов массива в языке C. Например, чтобы обратиться к верхнему левому элементу матрицы, следует написать:

```
D3DXMATRIX M;
M(0, 0) = 5.0f; // Присвоить первому элементу матрицы значение 5.0f.
```

Кроме того, библиотека D3DX предоставляет набор полезных функций, позволяющих инициализировать единичную матрицу **D3DXMATRIX**, транспонировать матрицу **D3DXMATRIX** и инвертировать матрицу **D3DXMATRIX**:

```
D3DXMATRIX *D3DXMatrixIdentity(
    D3DXMATRIX *pout      // Матрица, инициализируемая как единичная
  );

D3DXMATRIX M;
D3DXMatrixIdentity(&M);     // M = единичная матрица

D3DXMATRIX *D3DXMatrixTranspose(
    D3DXMATRIX *pOut,       // Результат транспонирования матрицы
    CONST D3DXMATRIX *pM    // Транспонируемая матрица
  );

D3DXMATRIX A(...);          // инициализация A
D3DXMATRIX B;
D3DXMatrixTranspose(&B, &A); // B = транспонированная A

D3DXMATRIX *D3DXMatrixInverse(
    D3DXMATRIX *pOut,      // возвращает результат инвертирования
    FLOAT *pDeterminant,   // детерминант, если необходим, иначе 0
    CONST D3DXMATRIX *pM   // инвертируемая матрица
  );
```

Функция инвертирования возвращает **NULL**, если переданная ей матрица не может быть инвертирована. Кроме того, в этой книге мы игнорируем второй параметр и всегда передаем в нем 0.

```
D3DXMATRIX A(...);            // инициализация A
D3DXMATRIX B;
D3DXMatrixInverse(&B, 0, &A); // B = инвертированная A
```

Основные преобразования

Создавая использующие Direct3D программы, для представления преобразований мы будем применять матрицы 4 × 4. Идея заключается в следующем: мы инициализируем элементы матрицы **X** размером 4 × 4 таким образом, чтобы они описывали требуемое преобразование. Затем мы помещаем координаты точки или компоненты вектора в столбцы вектора-строки **v** размером 1 × 4. Результатом произведения **vX** будет новый преобразованный вектор **v'**. Например, если матрица **X** представляет перемещение на 10 единиц вдоль оси X, и **v** = [2, 6, –3, 1], произведение **vX** = **v'** = [12, 6, –3, 1].

Следует пояснить несколько моментов. Мы используем матрицы размера 4 × 4 по той причине, что они позволяют представить все необходимые нам преобразования. На первый взгляд матрицы размером 3 × 3 кажутся более подходящими для трехмерной графики. Однако, с их помощью нельзя представить ряд преобразований, которые могут нам потребоваться, таких как перемещение, перспективная проекция и отражение. Помните, что мы работаем с произведением вектора на матрицу и при выполнении преобразований ограничены правилами умножения матриц. Дополнение матрицы до размера 4 × 4 позволяет нам с помощью матрицы описать большинство преобразований и при этом произведение вектора на матрицу будет определено.

Мы упомянули, что координаты точки или компоненты вектора будем хранить в столбцах вектора-строки размером 1 × 4. Но наши точки и векторы — трехмерные! Зачем же использовать вектор-строку 1 × 4? Мы должны дополнить наши трехмерные точки/векторы до четырехмерного вектора-строки 1 × 4 чтобы был определен результат умножения вектора на матрицу; произведение вектора-строки 1 × 3 и матрицы 4 × 4 *не определено*.

Так какое же значение использовать для четвертой компоненты, которую, кстати, мы будем обозначать *w*? Когда вектор-строка 1 × 4 используется для представления *точки*, значение *w* будет равно 1. Это позволяет корректно выполнять перемещение точки. Поскольку вектор не зависит от местоположения, операция перемещения векторов не определена и результат попытки переместить вектор не имеет смысла. Чтобы предотвратить перемещение векторов мы, помещая компоненты *вектора* в вектор-строку 1 × 4, присваиваем компоненте *w* значение 0. Например, точка **p** = (p_1, p_2, p_3), помещенная в вектор-строку 1 × 4 будет выглядеть как [p_1, p_2, p_3, 1], а вектор **v** = (v_1, v_2, v_3), помещенный в вектор-строку 1 × 4 будет выглядеть как [v_1, v_2, v_3, 0].

ПРИМЕЧАНИЕ Мы устанавливаем *w* = 1 чтобы корректно осуществлялось перемещение точек. Мы устанавливаем *w* = 0, чтобы предотвратить перемещение векторов. Это станет более ясно, когда мы рассмотрим реальную матрицу переноса.

ПРИМЕЧАНИЕ Дополненный четырехмерный вектор называется *однородным вектором* (*homogenous vector*) и, поскольку однородный вектор может описывать и точки и векторы, мы будем использовать термин «вектор», подразумевая, что он может относиться как к точке, так и к вектору.

Иногда в результате преобразований компонента w вектора будет меняться таким образом, что $w \neq 0$ и $w \neq 1$. Взгляните на следующий пример:

$$\mathbf{p} = \begin{bmatrix} p_1, & p_2, & p_3, & \mathbf{1} \end{bmatrix} \begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} = \begin{bmatrix} p_1, & p_2, & p_3, & p_3 \end{bmatrix} = \mathbf{p}'$$

для $p_3 \neq 0$ и $p_3 \neq 1$.

Обратите внимание, что $w = p_3$. Когда $w \neq 0$ и $w \neq 1$, мы говорим, что у нас есть вектор в *однородном пространстве* (*homogeneous space*), в противоположность векторам в трехмерном пространстве. Мы можем отобразить вектор в однородном пространстве обратно на трехмерное пространство, разделив каждую компоненту вектора на значение компоненты w. Например, для отображения вектора (x, y, z, w) в однородном пространстве в трехмерный вектор **x**, выполним следующие операции:

$$\left(\frac{x}{w}, \quad \frac{y}{w}, \quad \frac{z}{w}, \quad \frac{w}{w} \right) = \left(\frac{x}{w}, \quad \frac{y}{w}, \quad \frac{z}{w}, \quad \mathbf{1} \right) = \left(\frac{x}{w}, \quad \frac{y}{w}, \quad \frac{z}{w} \right) = \mathbf{x}$$

Переход к однородному пространству и обратное отображение векторов в трехмерное пространство используются в программировании трехмерной графики для выполнения перспективной проекции.

ПРИМЕЧАНИЕ Когда мы записываем точку (x, y, z) в виде $(x, y, z, 1)$ мы фактически описываем наше трехмерное пространство как плоскость в четырехмерном пространстве, а именно четырехмерную плоскость $w = 1$. (Обратите внимание, что плоскость в четырехмерном пространстве является трехмерным пространством, точно так же как плоскость в трехмерном пространстве является двухмерным пространством.) Таким образом, присваивая w какое-нибудь другое значение, мы перемещаемся с полскости $w = 1$. Чтобы вернуться на эту плоскость, которая соответствует нашему трехмерному пространству, мы выполняем обратную проекцию путем деления каждой компоненты на w.

Матрица перемещения

Мы можем переместить вектор $(x, y, z, 1)$ на p_x единиц по оси X, p_y единиц по оси Y и p_z единиц по оси Z умножив его на следующую матрицу:

$$\mathbf{T}(\mathbf{p}) = \begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ p_x & p_y & p_z & \mathbf{1} \end{bmatrix}$$

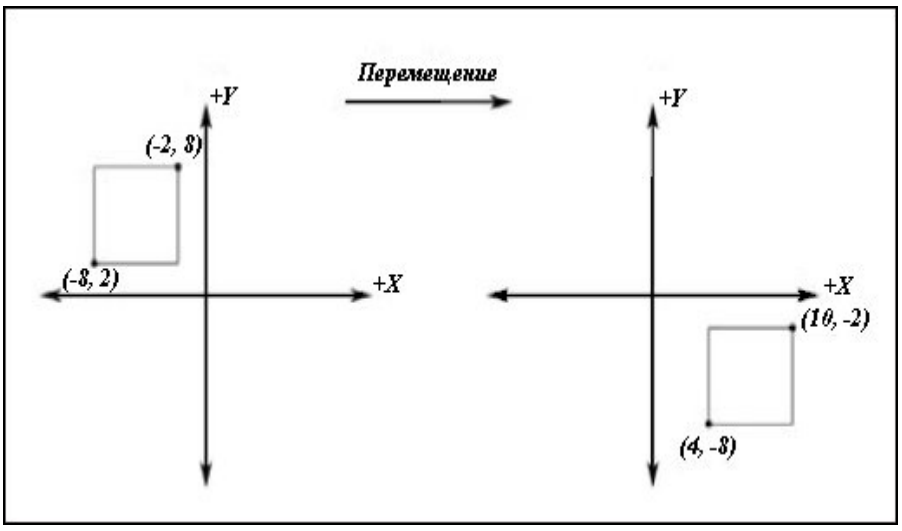

Рис. 8. *Перемещение на 12 единиц по оси X и на –10 единиц по оси Y*

Для создания матрицы перемещения в библиотеке D3DX используется следующая функция:

```
D3DXMATRIX *D3DXMatrixTranslation(
     D3DXMATRIX* pOut, // Результат
     FLOAT x,          // Количество единиц для перемещения по оси X
     FLOAT y,          // Количество единиц для перемещения по оси Y
     FLOAT z           // Количество единиц для перемещения по оси Z
);
```

УПРАЖНЕНИЕ

Пусть $\mathbf{T(p)}$ — это матрица, представляющая преобразование перемещения и пусть $\mathbf{v} = [v_1, v_2, v_3, 0]$ — это произвольный вектор. Проверьте, что $\mathbf{vT(p)} = \mathbf{v}$ (то есть, что если $w = 0$, преобразование перемещения не влияет на вектор).

Инверсия матрицы перемещения получается путем простой смены знака компонент вектора перемещения $\mathbf{p}$.

$$\mathbf{T}^{-1} = \mathbf{T}(-\mathbf{p}) = \begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ -p_x & -p_y & -p_z & \mathbf{1} \end{bmatrix}$$

Матрицы вращения

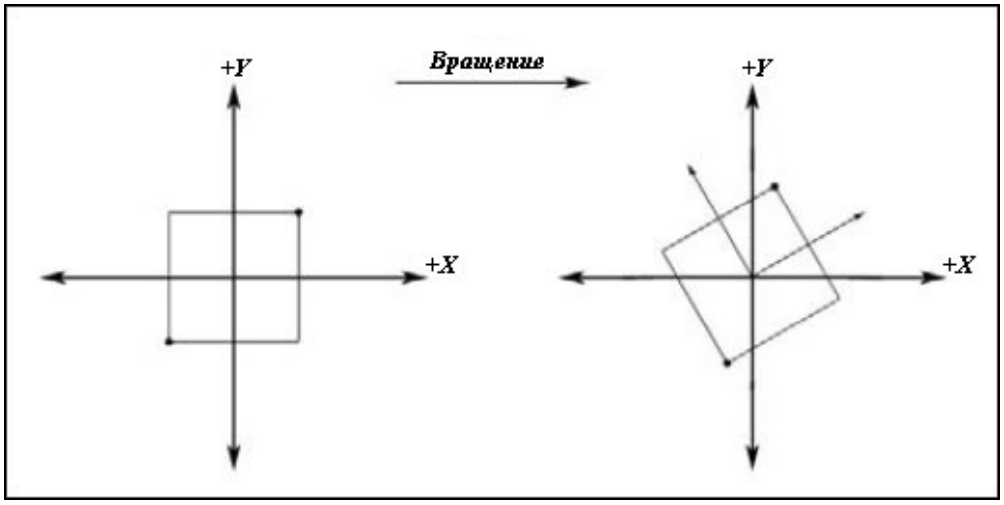

Рис. 9. *Поворот на 30 градусов против часовой стрелки вокруг оси Z*

Используя приведенные ниже матрицы мы можем повернуть вектор на φ радиан вокруг осей X, Y или Z. Обратите внимание, что если смотреть вдоль оси вращения по направлению к началу координат, то углы измеряются по часовой стрелке.

$$\mathbf{X}(\varphi)=\begin{bmatrix}\mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \cos(\varphi) & \sin(\varphi) & \mathbf{0}\\ \mathbf{0} & -\sin(\varphi) & \cos(\varphi) & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1}\end{bmatrix}$$

Для создания матрицы вращения вокруг оси X в библиотеке D3DX используется следующая функция:

```
D3DXMATRIX *D3DXMatrixRotationX(
     D3DXMATRIX* pOut, // Результат
     FLOAT Angle       // Угол поворота в радианах
);
```

$$\mathbf{Y}(\varphi)=\begin{bmatrix}\cos(\varphi) & \mathbf{0} & -\sin(\varphi) & \mathbf{0}\\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0}\\ \sin(\varphi) & \mathbf{0} & \cos(\varphi) & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1}\end{bmatrix}$$

Для создания матрицы вращения вокруг оси Y в библиотеке D3DX используется следующая функция:

```
D3DXMATRIX *D3DXMatrixRotationY(
     D3DXMATRIX* pOut, // Результат
     FLOAT Angle       // Угол поворота в радианах
);
```

$$\mathbf{Z}(\varphi)=\begin{bmatrix}\cos(\varphi) & \sin(\varphi) & \mathbf{0} & \mathbf{0}\\ -\sin(\varphi) & \cos(\varphi) & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1}\end{bmatrix}$$

Для создания матрицы вращения вокруг оси Z в библиотеке D3DX используется следующая функция:

```
D3DXMATRIX *D3DXMatrixRotationZ(
     D3DXMATRIX* pOut, // Результат
     FLOAT Angle       // Угол поворота в радианах
);
```

Инверсией матрицы вращения $\mathbf{R}$ является результат транспонирования этой матрицы: $\mathbf{R}^T=\mathbf{R}^{-1}$. Такие матрицы называются *ортогональными*.

Матрица масштабирования

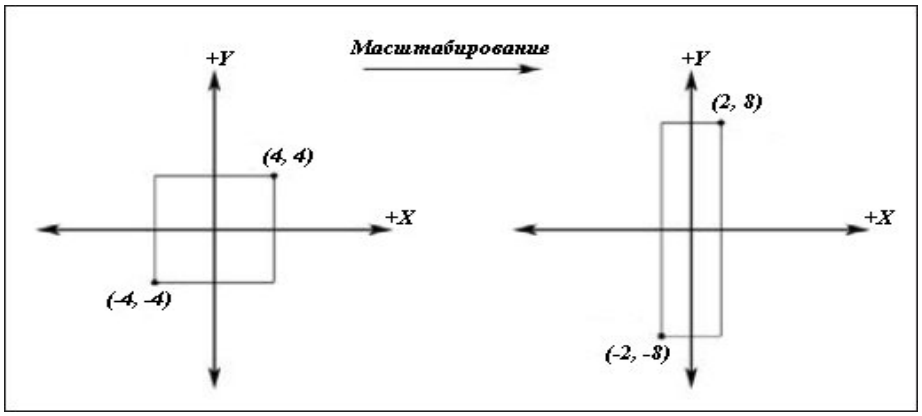

Рис. 10. *Масштабирование с коэффициентом 1/2 по оси X и коэффициентом 2 по оси Y*

Мы можем масштабировать вектор с коэффициентом q_x по оси X, коэффициентом q_y по оси Y и коэффициентом q_z по оси Z, умножив его на следующую матрицу:

$$\mathbf{S}(\mathbf{q})=\begin{bmatrix}q_x & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & q_y & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & q_z & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1}\end{bmatrix}$$

Для создания матрицы масштабирования в библиотеке D3DX используется следующая функция:

```
D3DXMATRIX *D3DXMatrixScaling(
     D3DXMATRIX* pOut, // Результат
     FLOAT sx,         // Коэффициент масштабирования по оси X
     FLOAT sy,         // Коэффициент масштабирования по оси Y
     FLOAT sz          // Коэффициент масштабирования по оси Z
);
```

Чтобы инвертировать матрицу масштабирования надо взять обратную дробь для каждого коэффициента масштабирования:

$$\mathbf{S}^{-1} = \mathbf{S}\left(\frac{1}{q_x}, \frac{1}{q_y}, \frac{1}{q_z}\right) = \begin{bmatrix} \frac{1}{q_x} & 0 & 0 & 0 \\ 0 & \frac{1}{q_y} & 0 & 0 \\ 0 & 0 & \frac{1}{q_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Комбинирование преобразований

Часто мы будем применять к векторам целую последовательность преобразований. Например, мы можем масштабировать вектор, затем повернуть его и потом переместить в требуемую позицию.

В качестве примера мы рассмотрим вектор $\mathbf{p} = [5, 0, 0, 1]$, который масштабируем по всем осям с коэффициентом 1/5, затем повернем его на $\pi/4$ радиан вокруг оси Y и, наконец, переместим на 1 единицу по оси X, 2 единицы по оси Y и –3 единицы по оси Z.

Обратите внимание, что мы должны выполнить масштабирование, поворот вокруг оси Y и перемещение. Мы инициализируем наши матрицы преобразований $\mathbf{S}$, $\mathbf{R}_y$, $\mathbf{T}$ для масштабирования, поворота и перемещения соответственно, следующим образом:

$$\mathbf{S}\left(\frac{1}{5}, \frac{1}{5}, \frac{1}{5}\right) = \begin{bmatrix} \frac{1}{5} & 0 & 0 & 0 \\ 0 & \frac{1}{5} & 0 & 0 \\ 0 & 0 & \frac{1}{5} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R}_y\left(\frac{\pi}{4}\right) = \begin{bmatrix} 0.707 & 0 & -0.707 & 0 \\ 0 & 1 & 0 & 0 \\ 0.707 & 0 & 0.707 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{T}(1,\ \ 2,\ \ -3) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 2 & -3 & 1 \end{bmatrix}$$

Применив последовательность преобразований в заданном порядке — масштабирование, вращение, перемещение — получим:

$$\begin{aligned} &\mathbf{pS} = [1,\ \ 0,\ \ 0,\ \ 1] = \mathbf{p}' \\ (5)\quad &\mathbf{p'R}_y = [0.707,\ \ 0,\ \ -0.707,\ \ 1] = \mathbf{p}'' \\ &\mathbf{p}''\mathbf{T} = [1.707,\ \ 2,\ \ -3.707\ \ 1] \end{aligned}$$

Ключевым преимуществом использования матриц является возможность использования умножения матриц для комбинирования нескольких преобразований в одной матрице. Вернемся к примеру, рассматриваемому в начале данного раздела. Давайте с помощью умножения матриц скомбинируем все три матрицы преобразований в одну, которая будет представлять все три преобразования сразу. Обратите внимание, что порядок, в котором мы умножаем матрицы должен соответствовать порядку применения отдельных преобразований.

$$(6)\quad \mathbf{SR}_y\mathbf{T} = \begin{bmatrix} \frac{1}{5} & 0 & 0 & 0 \\ 0 & \frac{1}{5} & 0 & 0 \\ 0 & 0 & \frac{1}{5} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.707 & 0 & -0.707 & 0 \\ 0 & 1 & 0 & 0 \\ 0.707 & 0 & 0.707 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 2 & -3 & 1 \end{bmatrix} =$$

$$= \begin{bmatrix} 0.1414 & 0 & -0.1414 & 0 \\ 0 & 1 & 0 & 0 \\ 0.1414 & 0 & 0.1414 & 0 \\ 1 & 2 & -3 & 1 \end{bmatrix} = \mathbf{Q}$$

В результате получаем $\mathbf{pQ}$ = [1.707, 2, –3.707, 1].

Возможность комбинировать преобразования значительно увеличивает производительность. Представьте, что вам необходимо применить одну и ту же последовательность преобразований масштабирования, вращения и перемещения к большому набору векторов (это обычная задача в трехмерной компьютерной графике). Вместо того, чтобы применять к каждому вектору последовательность

преобразований, как мы делали это в формуле (5), можно скомбинировать все три преобразования в одной матрице, как мы делали это в формуле (6). После этого остается только умножить каждый вектор на единственную матрицу, содержащую комбинацию всех трех преобразований. Благодаря этому количество выполняемых операций умножения вектора на матрицу значительно сокращается.

Некоторые функции для преобразования векторов

Библиотека D3DX предоставляет две функции для преобразования точек и векторов соответственно. Функция **D3DXVec3TransformCoord** используется для преобразования точек и предполагает, что четвертая компонента вектора равна 1. Функция **D3DXVec3TransformNormal** используется для преобразования векторов и предполагает, что четвертая компонента вектора равна 0.

```
D3DXVECTOR3 *D3DXVec3TransformCoord(
     D3DXVECTOR3* pOut,     // Результат
     CONST D3DXVECTOR3* pV, // Преобразуемая точка
     CONST D3DXMATRIX* pM   // Матрица преобразования
);

D3DXMATRIX T(...);          // инициализация матрицы преобразований
D3DXVECTOR3 p(...);         // инициализация точки
D3DXVec3TransformCoord(&p, &p, &T); // преобразование точки

D3DXVECTOR3 *D3DXVec3TransformNormal(
     D3DXVECTOR3 *pOut,     // Результат
     CONST D3DXVECTOR3 *pV, // Преобразуемый вектор
     CONST D3DXMATRIX *pM   // Матрица преобразования
);

D3DXMATRIX T(...);          // инициализация матрицы преобразований
D3DXVECTOR3 v(...);         // инициализация вектора
D3DXVec3TransformNormal(&v, &v, &T); // преобразование вектора
```

ПРИМЕЧАНИЕ Библиотека D3DX также предоставляет функции **D3DXVec3TransformCoordArray** и **D3DXVec3Transform-NormalArray** для преобразования массива точек и массива векторов соответственно.

Плоскости

Плоскость описывается с помощью вектора $\mathbf{n}$ и принадлежащей плоскости точки $\mathbf{p}_0$. Вектор $\mathbf{n}$ называется *вектором нормали* (*normal vector*) плоскости и должен быть перпендикулярен плоскости (рис. 11).

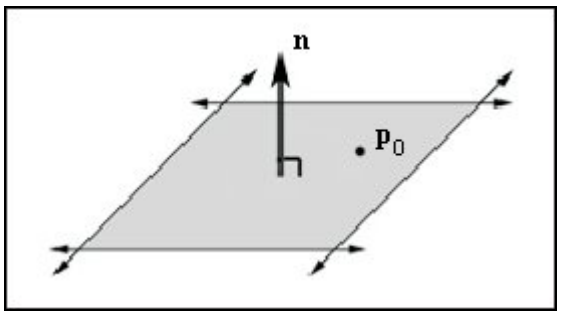

Рис. 11. *Плоскость, заданная вектором нормали* ***n*** *и точкой плоскости* $\mathbf{p}_0$

На рис. 12 мы видим, что графическим представлением плоскости является множество всех точек, удовлетворяющих условию

$$(7) \quad \mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = \mathbf{0}$$

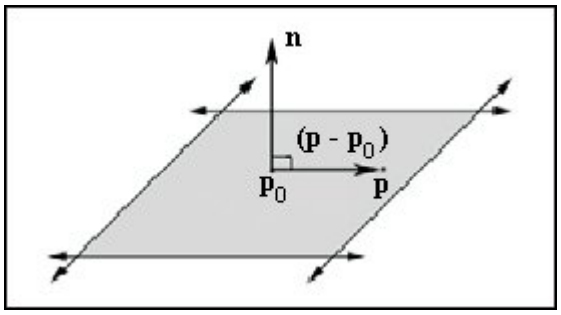

Рис. 12. *Если точка* $\mathbf{p}_0$ *принадлежит плоскости, то точка* ***p*** *также принадлежит этой плоскости в том случае, если вектор* $(\mathbf{p} - \mathbf{p}_0)$ *перпендикулярен вектору нормали плоскости*

При описании конкретной плоскости вектор нормали **n** и принадлежащая плоскости точка $\mathbf{p}_0$ обычно фиксированы, и формула (7) записывается в следующем виде:

$$(8) \quad \mathbf{n} \cdot \mathbf{p} + d = \mathbf{0}$$

где $d = -\mathbf{n} \cdot \mathbf{p}_0$.

ПРИМЕЧАНИЕ	Если длина вектора нормали плоскости **n** равна единице, $d = -\mathbf{n} \cdot \mathbf{p}_0$ — это наименьшее расстояние от начала координат до плоскости.

D3DXPLANE

Для представления плоскости в коде достаточно указать вектор нормали **n** и константу *d*. Можно думать об этом как о четырехмерном векторе, который мы будем обозначать $(\mathbf{n}, d)$. В библиотеке D3DX для плоскостей используется следующая структура:

```
typedef struct D3DXPLANE
{
#ifdef __cplusplus
public:
    D3DXPLANE() {}
    D3DXPLANE(CONST FLOAT*);
    D3DXPLANE(CONST D3DXFLOAT16*);
    D3DXPLANE(FLOAT a, FLOAT b, FLOAT c, FLOAT d);

    // приведение типа
    operator FLOAT* ();
    operator CONST FLOAT* () const;

    // унарные операторы
    D3DXPLANE operator + () const;
    D3DXPLANE operator - () const;

    // бинарные операторы
    BOOL operator == (CONST D3DXPLANE&) const;
    BOOL operator != (CONST D3DXPLANE&) const;
#endif //__cplusplus
    FLOAT a, b, c, d;
} D3DXPLANE, *LPD3DXPLANE;
```

где **a**, **b**, и **c** — это компоненты вектора нормали плоскости **n**, а **d** — это константа *d* из формулы (8).

Взаимное расположение точки и плоскости

Формула (8) в основном используется для проверки местоположения точки относительно плоскости. Предположим, нам дана плоскость (**n**, *d*), и мы хотим узнать как точка **p** расположена относительно этой плоскости:

- Если $\mathbf{n} \cdot \mathbf{p} + d = 0$, то точка **p** принадлежит плоскости.
- Если $\mathbf{n} \cdot \mathbf{p} + d > 0$, то точка **p** находится перед плоскостью в положительном полупространстве плоскости.
- Если $\mathbf{n} \cdot \mathbf{p} + d < 0$, то точка **p** находится за плоскостью в отрицательном полупространстве плоскости.

ПРИМЕЧАНИЕ Если длина вектора нормали плоскости **n** равна единице, $\mathbf{n} \cdot \mathbf{p} + d$ — это наименьшее расстояние от плоскости до точки **p**.

Приведенная ниже функция библиотеки D3DX вычисляет значение $\mathbf{n} \cdot \mathbf{p} + d$ для заданных плоскости и точки:

```
FLOAT D3DXPlaneDotCoord(
    CONST D3DXPLANE *pP,        // плоскость
    CONST D3DXVECTOR3 *pV       // точка
);
```

```
// Проверка местоположения точки относительно плоскости
D3DXPLANE p(0.0f, 1.0f, 0.0f, 0.0f);

D3DXVECTOR3 v(3.0f, 5.0f, 2.0f);

float x = D3DXPlaneDotCoord(&p, &v);

if( x приблизительно равно 0.0f ) // v принадлежит плоскости
if( x > 0 )            // v в положительном полупространстве
if( x < 0 )            // v в отрицательном полупространстве
```

ПРИМЕЧАНИЕ Мы говорим «приблизительно равно», чтобы напомнить об особенностях сравнения чисел с плавающей точкой. Обратите внимание на примечание в разделе «Равенство векторов».

ПРИМЕЧАНИЕ Существуют похожие на **D3DXPlaneDotCoord** методы **D3DXPlaneDot** и **D3DXPlaneDotNormal**. Их описание можно посмотреть в документации DirectX.

Создание плоскостей

Вместо непосредственного указания нормали и кратчайшего расстояния до начала координат, мы можем использовать еще два способа задания плоскостей. Зная вектор нормали $\mathbf{n}$ и принадлежащую плоскости точку $\mathbf{p}_0$ мы можем вычислить значение d следующим образом:

$$\mathbf{n} \cdot \mathbf{p}_0 + d = \mathbf{0}$$
$$\mathbf{n} \cdot \mathbf{p}_0 = -d$$
$$-\mathbf{n} \cdot \mathbf{p}_0 = d$$

Для выполнения этих вычислений библиотека D3DX предоставляет следующую функцию:

```
D3DXPLANE *D3DXPlaneFromPointNormal(
    D3DXPLANE* pOut,              // Результат
    CONST D3DXVECTOR3* pPoint,    // Точка плоскости
    CONST D3DXVECTOR3* pNormal    // Нормаль плоскости
);
```

Кроме того, мы можем создать плоскость, указав три принадлежащие ей точки.

Имея три точки $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$, мы можем сформировать два вектора плоскости:

$$\mathbf{u} = \mathbf{p}_1 - \mathbf{p}_0$$
$$\mathbf{v} = \mathbf{p}_2 - \mathbf{p}_0$$

Затем с помощью векторного произведения можно вычислить нормаль плоскости. Помните о правиле левой руки.

$$\mathbf{n} = \mathbf{u} \times \mathbf{v}$$

Следовательно, $-(\mathbf{n} \cdot \mathbf{p}_0) = d$.

Для создания плоскости по трем заданным точкам библиотека D3DX предоставляет следующую функцию:

```
D3DXPLANE *D3DXPlaneFromPoints(
    D3DXPLANE* pOut,          // Результат
    CONST D3DXVECTOR3* pV1,   // Первая точка плоскости
    CONST D3DXVECTOR3* pV2,   // Вторая точка плоскости
    CONST D3DXVECTOR3* pV3    // Третья точка плоскости
);
```

Нормализация плоскости

Иногда может сложиться такая ситуация, что у нас есть плоскость и нам надо нормализовать ее вектор нормали. На первый взгляд нам достаточно нормализовать вектор нормали как любой другой вектор. Но вспомните, что в формуле $\mathbf{n} \cdot \mathbf{p} + d = 0$ $d = -\mathbf{n} \cdot \mathbf{p}_0$. Как видите, длина вектора нормали влияет на константу d. Следовательно, если мы нормализуем вектор нормали, нам надо заново вычислить d. Обратите внимание, что

$$\frac{d}{|\mathbf{n}|} = -\frac{\mathbf{n}}{|\mathbf{n}|} \cdot \mathbf{p}_0$$

Следовательно, для нормализации вектора нормали плоскости ($\mathbf{n}$, d) мы получаем следующую формулу:

$$\frac{\mathbf{1}}{|\mathbf{n}|}(\mathbf{n}, \quad d) = \left(\frac{\mathbf{n}}{|\mathbf{n}|}, \quad \frac{d}{|\mathbf{n}|}\right)$$

Для нормализации вектора нормали плоскости можно использовать следующую функцию библиотеки D3DX:

```
D3DXPLANE *D3DXPlaneNormalize(
    D3DXPLANE *pOut,        // Нормализованная плоскость
    CONST D3DXPLANE *pP     // Исходная плоскость
);
```

Преобразование плоскости

Ленджел в своей книге «Mathematics for 3D Game Programming & Computer Graphics» показал, что мы можем преобразовать плоскость ($\hat{\mathbf{n}}$, d) представив ее как четырехмерный вектор и умножив на результат транспонирования инверсии описывающей желаемое преобразование матрицы. Обратите внимание, что перед этим необходимо нормализовать вектор нормали плоскости.

Для выполнения этих действий можно использовать следующую функцию библиотеки D3DX:

```
D3DXPLANE *D3DXPlaneTransform(
    D3DXPLANE *pOut,          // Результат
    CONST D3DXPLANE *pP,      // Исходная плоскость
    CONST D3DXMATRIX *pM      // Матрица преобразования
);
```

А вот пример кода:

```
D3DXMATRIX T(...);     // Инициализация. T - требуемое преобразование
D3DXMATRIX inverseOfT;
D3DXMATRIX inverseTransposeOfT;

D3DXMatrixInverse(&inverseOfT, 0, &T);
D3DXMatrixTranspose(&inverseTransposeOfT, &inverseOfT);

D3DXPLANE p(...);             // Инициализация плоскости
D3DXPlaneNormalize(&p, &p);   // Нормализация вектора нормали
D3DXPlaneTransform(&p, &p, &inverseTransposeOfT);
```

Точка плоскости, ближайшая к заданной

Пусть у нас есть точка **p** в трехмерном пространстве и нам необходимо найти точку **q**, принадлежащую плоскости ($\hat{\mathbf{n}}$, d) и ближайшую к точке **p**. Обратите внимание: предполагается что вектор нормали плоскости нормализован, это упрощает решение задачи.

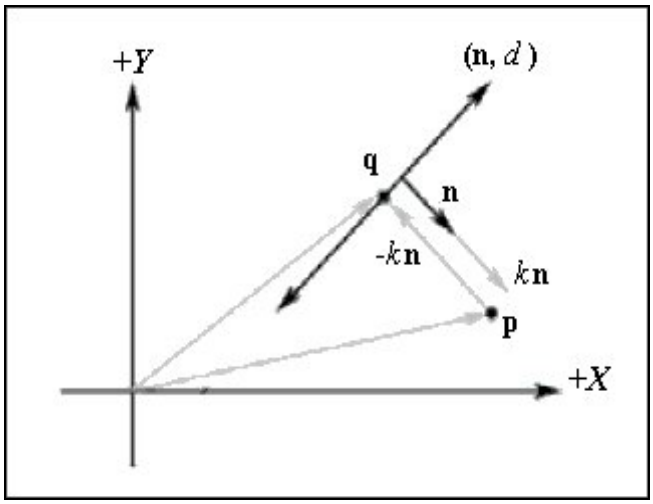

Рис. 13. *Точка **q** плоскости ($\hat{\mathbf{n}}$, d) ближайшая к точке **p**. Обратите внимание, что кратчайшее расстояние k от точки **p** до плоскости положительно, если точка **p** находится в положительном полупространстве плоскости ($\hat{\mathbf{n}}$, d). Если же точка **p** находится за плоскостью, то $k < 0$*

На рис. 13 видно, что $\mathbf{q} = \mathbf{p} + (-k\hat{\mathbf{n}})$, где k — это кратчайшее расстояние от точки **p** до плоскости, которое одновременно является и кратчайшим расстоянием между точками **p** и **q**. Помните, что если вектор нормали плоскости **n** нормализован, то $\mathbf{n} \cdot \mathbf{p} + d$ является кратчайшим расстоянием от плоскости до точки **p**.

Лучи

Предположим, что в разрабатываемой нами игре игрок стреляет из ружья в противника. Как определить попала ли в цель пуля, выпущенная из заданной точки в указанном направлении? Один из возможных подходов: моделирование траектории пули с помощью луча и моделирование врага с помощью *ограничивающей сферы* (*bounding sphere*). (Ограничивающая сфера — это просто сфера минимального диаметра, в которую помещается весь объект целиком, что позволяет приблизительно представить занимаемый им объем. Более подробно об ограничивающих сферах мы поговорим в главе 11.) Тогда с помощью математических вычислений мы можем определить пересекает ли луч сферу и, если да, то где. В данном разделе мы обсудим математическую модель лучей.

Лучи

Луч описывается путем указания начальной точки и направления. Параметрическая формула луча выглядит следующим образом:

$$(9) \quad \mathbf{p}(t) = \mathbf{p}_0 + t\mathbf{u}$$

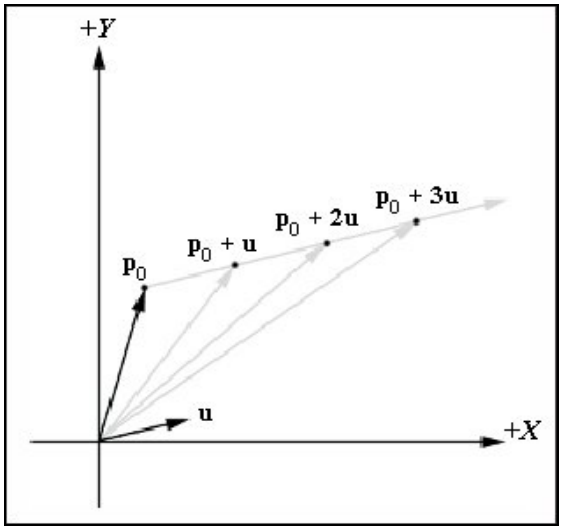

Рис. 14. *Луч, заданный начальной точкой* $\mathbf{p}_0$ *и вектором направления* $\mathbf{u}$*. Мы можем генерировать точки луча, подставляя в формулу различные значения t, которые должны быть больше или равны нулю*

В формуле луча $\mathbf{p}_0$ — это начальная точка, $\mathbf{u}$ — это вектор, задающий направление луча, а t — это параметр. Подставляя различные значения t, мы сможем получать координаты различных точек луча. Причем для луча значение t должно находиться в диапазоне $[0, \infty)$. Значения меньше нуля приведут к вычислению координат точек, находящихся за лучом (на прямой, частью которой является луч). Фактически, если t принимает значения из диапазона $(-\infty, \infty)$, мы получаем линию в трехмерном пространстве.

Пересечение луча и плоскости

Предположим, у нас есть луч $\mathbf{p}(t) = \mathbf{p}_0 + t\mathbf{u}$ и плоскость $\mathbf{n} \cdot \mathbf{p} + d = 0$, и мы хотим определить пересекает ли луч плоскость и, если да, то вычислить координаты точки пересечения. Для этого мы помещаем формулу луча в формулу плоскости и вычисляем такое значение параметра t, которое удовлетворяет уравнению плоскости. Подстановка найденного значения в уравнение луча позволяет вычислить координаты точки пересечения.

Подставляем формулу (9) в формулу плоскости:

$\mathbf{n} \cdot \mathbf{p}(t) + d = \mathbf{0}$ Подставляем уравнение луча в формулу плоскости.

$\mathbf{n} \cdot (\mathbf{p}_0 + t\mathbf{u}) + d = \mathbf{0}$

$\mathbf{n} \cdot \mathbf{p}_0 + \mathbf{n} \cdot t\mathbf{u} + d = \mathbf{0}$ Раскрываем скобки.

$\mathbf{n} \cdot t\mathbf{u} = -d - (\mathbf{n} \cdot \mathbf{p}_0)$

$t(\mathbf{n} \cdot \mathbf{u}) = -d - (\mathbf{n} \cdot \mathbf{p}_0)$ Выносим за скобки переменную.

$t = \dfrac{-d - (\mathbf{n} \cdot \mathbf{p}_0)}{(\mathbf{n} \cdot \mathbf{u})}$ Решение для t.

Если значение t не находится в диапазоне $[0, \infty)$, значит луч не пересекает плоскость.

Если значение t находится в диапазоне $[0, \infty)$, точка пересечения находится путем подстановки найденного значения параметра в формулу луча:

$$\mathbf{p}\left(\frac{-d - (\mathbf{n} \cdot \mathbf{p}_0)}{(\mathbf{n} \cdot \mathbf{u})}\right) = \mathbf{p}_0 + \frac{-d - (\mathbf{n} \cdot \mathbf{p}_0)}{(\mathbf{n} \cdot \mathbf{u})}\mathbf{u}$$

Итоги

- Векторы используются для моделирования физических величин, которые характеризуются величиной и направлением. Геометрическим представлением вектора является направленный отрезок прямой. Когда вектор находится в стандартной позиции его начало совпадает с началом координат. Вектор в стандартной позиции описывается путем указания координат конца вектора.
- Мы можем использовать матрицы 4×4 для представления преобразований и однородные векторы 1×4 для описания точек и векторов. В результате умножения вектора-строки 1×4 на матрицу преобразования 4×4 получается новый преобразованный вектор-строка 1×4. Можно скомбинировать несколько матриц преобразований в одну перемножив их друг на друга.

- Для представления векторов и точек мы используем однородные четырехмерные векторы. Для вектора значение компоненты w равно 0, а для точки значение компоненты w равно 1. Если $w \neq 0$ и $w \neq 1$, то у нас есть вектор (x, y, z, w) в однородном пространстве, который может быть отображен обратно на трехмерное пространство путем деления каждой его компоненты на w;

$$\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, \frac{w}{w}\right) = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, \mathbf{1}\right) = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$$

- Плоскости делят трехмерное пространство на две части: положительное полупространство перед плоскостью и отрицательное полупространство за ней. Плоскости применяются для проверки местоположения точек относительно них (другими словами, чтобы проверить в каком полупространстве относительно данной плоскости находится точка).
- Лучи описываются путем указания начальной точки и вектора направления. Лучи полезны для моделирования различных физических явлений, таких как луч света или полет снаряда (например, пули или ракеты) по прямолинейной траектории.

Часть II
Основы Direct3D

В этой части вы изучите основные концепции и техники Direct3D, которые будете применять в остальной части книги. Овладев этими начальными знаниями можно перейти к написанию более интересных приложений. Ниже приведен краткий обзор, входящих в эту часть книги глав.

Глава 1, «Инициализация Direct3D» — В этой главе вы узнаете что такое Direct3D и как инициализировать его, чтобы подготовиться к рисованию трехмерной графики.

Глава 2, «Конвейер визуализации» — Первой темой этой главы является изучение того, как посредством математики можно описать трехмерный мир и виртуальную камеру, представляющую собой точку из которой мы смотрим на мир. Второй темой является изучение этапов, необходимых для получения двухмерной картинки трехмерного мира, показывающей то, что «видит» камера; все эти этапы вместе называются *конвейером визуализации* (*rendering pipeline*).

Глава 3, «Рисование в Direct3D» — В данной главе мы покажем вам как рисовать трехмерные объекты в Direct3D. Вы узнаете как хранить геометрические данные в форме, пригодной для использования в Direct3D, и познакомитесь с командами рисования Direct3D. Кроме того, вы узнаете как с помощью состояний визуализации можно настраивать способ отображения объектов в Direct3D.

Глава 4, «Цвет» — В данной главе мы узнаем о том, как в Direct3D представляются цвета и как можно окрасить трехмерный графический примитив. Кроме того, мы рассмотрим два способа, какими назначенные вершинам цвета влияют на закрашивание всего примитива.

Глава 5, «Освещение» — В главе мы узнаем как создавать источники света и задавать взаимодействие световых лучей и поверхностей. Освещение добавляет сцене реализма и позволяет подчеркнуть форму и объем составляющих сцену объектов.

Глава 6, «Текстурирование» — В этой главе описывается *наложение текстур* (*texture mapping*). Это техника, используемая для увеличения реализма сцен путем наложения двухмерных изображений на трехмерные примитивы. Например, наложение текстур позволяет изобразить кирпичную стену наложив двухмерное изображение кирпичной стены на трехмерный прямоугольник.

Глава 7, «Смешивание» — В этой главе мы изучим технику, называемую *смешиванием* (*blending*). Она позволяет реализовать ряд эффектов, в частности, прозрачные объекты, которые будут выглядеть как стеклянные.

Глава 8, «Трафареты» — В этой главе рассматривается буфер трафарета, который позволяет указать, какие пикселы отображать, а какие — нет. Для иллюстрации рассматриваемых в главе идей мы рассмотрим реализацию с помощью буфера трафарета отражений и плоских теней.

Глава 1
Инициализация Direct3D

Исторически сложилось, что инициализация Direct3D была нудным и сложным процессом. К счастью, в версию 8.0 была добавлена упрощенная модель инициализации, и Direct3D 9.0 следует той же самой модели. Тем не менее, процесс инициализации подразумевает, что программист знаком с базовыми концепциями компьтерной графики и некоторыми основными типами данных Direct3D. Первые несколько разделов этой главы помогут вам соответствовать выдвигаемым требованиям. В оставшейся после подготовки части главы исследуется процесс инициализации.

Цели

- Изучить как Direct3D взаимодействует с графическим оборудованием.
- Понять, какую роль в Direct3D играет COM.
- Изучить основные концепции компьютерной графики, такие как хранение двухмерных изображений в памяти, переключение страниц и буфер глубины.
- Узнать, как инициализировать Direct3D.
- Познакомиться с общей структурой приложения, которая будет использоваться во всех примерах этой книги.

1.1 Обзор Direct3D

Direct3D — это низкоуровневый графический API (программный интерфейс для приложений), позволяющий отображать трехмерные миры используя аппаратные ускорители трехмерной графики. Direct3D можно представлять как посредника между приложением и графическим устройством (аппаратурой трехмерной графики). Например, чтобы приказать графическому устройству очистить экран, приложение должно вызвать метод Direct3D **`IDirect3DDevice9::Clear`**. Взаимоотношения между приложением, Direct3D и аппаратурой компьютера показаны на рис. 1.1.

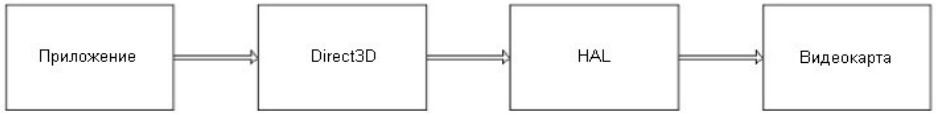

Рис. 1.1. *Взаимосвязь между приложением, Direct3D и аппаратурой*

На рис. 1.1 блок с названием Direct3D представляет набор документированных интерфейсов и функций, которые Direct3D предоставляет приложениям и программистам. Эти интерфейсы и функции охватывают полный набор функциональных возможностей, предлагаемых данной версией Direct3D. Обратите внимание, что предложение возможости Direct3D не означает, что она будет поддерживаться аппаратурой.

На рис. 1.1 изображена промежуточная стадия между Direct3D и графическим устройством — уровень абстрагирования от аппаратуры (Hardware Abstraction Layer, HAL). Direct3D не может напрямую взаимодействовать с аппаратурой, поскольку продаются сотни различных видеокарт и каждая видеокарта отличается набором поддерживаемых функций и способом реализации тех функций, которые поддерживаются. Например, две разные видеокарты могут совершенно по-разному выполнять очистку экрана. Поэтому Direct3D требует, чтобы производители оборудования реализовали уровень абстрагирования от оборудования (HAL), который представляет собой зависящий от аппаратуры код, указывающий устройству ка выполнять те или иные операции. Благодаря этому Direct3D не требуется знать специфику конкретных устройств, и его спецификации не зависят от используемого оборудования.

Производители видеокарт встраивают поддержку всех предлагаемых их оборудованием возможностей в HAL. Те возможности, которые предлагает Direct3D, но которые не поддерживает устройство, в HAL не реализованы. Попытка использования тех возможностей Direct3D, которые не реализованы в HAL приводит к ошибке, за исключением тех случаев, когда требуемая функция может быть воспроизведена программно, как , например, программная обработка вершин в библиотеке времени выполнения Direct3D. Так что, используя экзотические возможности, поддерживаемые лишь несколькими устройствами, предусмотрите в программе проверку поддержки видеокартой этих функций (подробнее об этом мы поговорим в разделе 1.3.8).

1.1.1 Устройство REF

Вам может потребоваться написать прогамму Direct3D, использующую те аппаратные возможности, которые не поддерживает ваша система. Для этих целей Direct3D предоставляет вспомогательный растеризатор (reference rasterizer, известный также как устройство REF), который программно эмулирует все предлагаемые API Direct3D возможности. Это позволяет вам писать и проверять код, использующий возможности Direct3D, не поддерживаемые вашей аппаратурой. Например, в четвертой части этой книги мы будем использовать вершинные и пиксельные шейдеры, которые могут не поддерживаться вашей видеокартой. Если ваша видеокарта не поддерживает шейдеры, вы можете проверить работу кода с помощью устройства REF. Очень важно понимать, что устройство REF предназначено только для разработчиков. Оно присутствует только в DirectX SDK и не может быть предоставлено конечным пользователям. Кроме того, следует помнить, что устройство REF работает очень медленно и не годится ни для каких целей, кроме тестирования.

1.1.2 D3DDEVTYPE

В коде устройство HAL указывается константой **`D3DDEVTYPE_HAL`**, являющейся членом перечисления **`D3DDEVTYPE`**. точно так же устройство REF указывается с помощью константы **`D3DDEVTYPE_REF`**, являющейся членом того же перечисления. Очень важно помнить эти типы устройств, поскольку при создании устройства вам придется указывать, какой именно тип использовать.

1.2 COM

Модель компонентных объектов (Component Object Model, COM) — это технология, позволяющая DirectX быть независимым от языка программирования и совместимым со всеми предыдущими версиями. Обычно мы будем ссылаться на COM-объект, как на интерфейс, о котором, в нашем случае, можно думать как о классе C++. Большинство особенностей COM при программировании для DirectX на C++ остаются прозрачными и никак не влияют на работу. Есть только один важный момент, о котором следует помнить: для получения указателя на COM-интерфейс необходимо вызвать специальную функцию или метод другого COM-интерфейса; нельзя пользоваться ключевым словом C++ **`new`**. Кроме того, завершив работу с COM-интерфейсом, следует вызвать его метод **`Release`** (все COM-интерфейсы наследуют функциональность от интерфеса **`IUnknown`**, в котором есть метод **`Release`**), а не удалять его оператором **`delete`**. COM-объекты самостоятельно осуществляют управление памятью.

Конечно, можно еще много говорить о COM, но эта информация не требуется для эффективного использования DirectX.

ПРИМЕЧАНИЕ В коде для обозначения COM-интерфейсов используется заглавная буква **`I`**. Например, COM-интерфейс, представляющий поверхность называется **`IDirect3DSurface9`**.

1.3 Предварительная подготовка

Процесс инициализации Direct3D требует знакомства с рядом основных концепций компьютерной графики и базовыми типами Direct3D. В данном разделе мы познакомимся с этими идеями и типами, чтобы в дальнейшем сосредоточиться на обсуждении инициализации Direct3D.

1.3.1 Поверхности

Поверхность (*surface*) — это прямоугольный массив пикселей, используемый Direct3D в основном для хранения двухмерных изображений. Основные параметры поверхности представлены на рис. 1.2. Обратите внимание, что хотя мы представляем данные поверхности в виде матрицы, в действительности данные пикселей хранятся в линейном массиве.

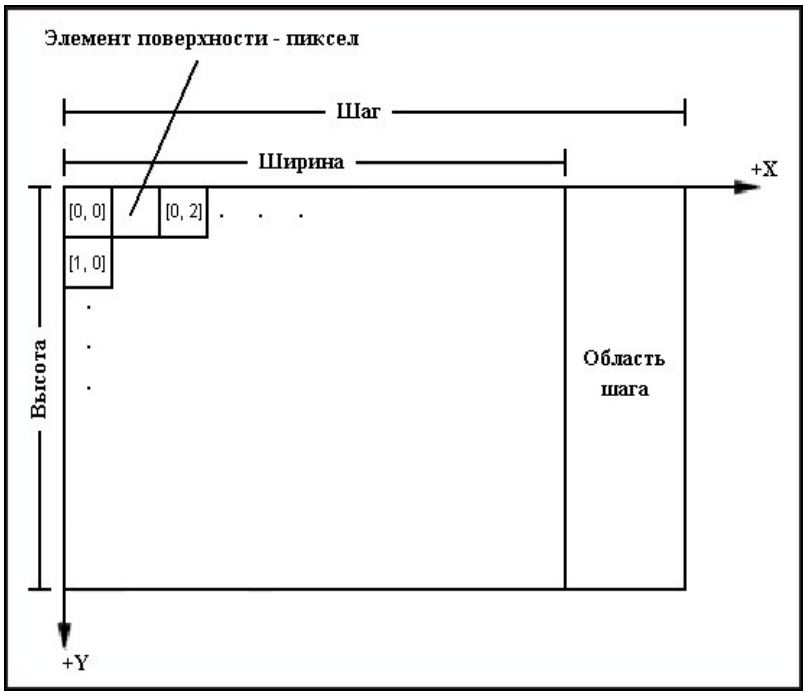

Рис. 1.2. *Поверхность*

Ширина и высота поверхности измеряются в пикселах. Шаг (pitch) поверхности измеряется в байтах. Более того, шаг поверхности может отличаться от ее ширины в зависимости от используемого оборудования. Вы *не должны* считать, что *шаг* = *ширина* * **sizeof(***пиксель***)**.

В коде поверхность представляется интерфейсом **`IDirect3DSurface9`**. Он предоставляет методы для прямого чтения и записи данных поверхности, а также методы для получения информации о поверхности. Вот наиболее важные методы интерфейса **`IDirect3DSurface9`**:

- **LockRect** — Этот метод позволяет получить указатель на память поверхности. После этого, используя арифметику указателей, мы можем читать и записывать отдельные пиксели поверхности.
- **UnlockRect** — После того, как мы вызвали метод **LockRect** и завершили работу с памятью поверхности, нам следует разблокировать поверхность, вызвав этот метод.
- **GetDesc** — Метод возвращает параметры поверхности, заполняя структуру **D3DSURFACE_DESC**.

Принимая во внимание наличие такого параметра, как шаг поверхности, блокировка поверхности и запись отдельных пикселей могут вызвать затруднения. Поэтому мы приводим фрагмент кода в котором поверхность блокируется и заполняется пикселями красного цвета:

```
// Предполагается, что _surface - указатель на интерфейс
// IDirect3DSurface9
// Предполагается, что у поверхности 32-разрядный формат пикселей

// Получаем описание поверхности
D3DSURFACE_DESC surfaceDesc;
_surface->GetDesc(&surfaceDesc);

// Получаем указатель на данные пикселей поверхности
D3DLOCKED_RECT lockedRect;
_surface->LockRect(
      &lockedRect,    // получаемый указатель на данные
      0,              // блокируем всю поверхность
      0);             // дополнительных параметров блокировки нет

// Перебираем все пиксели поверхности и делаем их красными
DWORD* imageData = (DWORD*)lockedRect.pBits;
for(int i = 0; i < surfaceDesc.Height; i++)
{
    for(int j = 0; j < surfaceDesc.Width; j++)
    {
        // индекс в текстуре, обратите внимание, что мы
        // используем шаг и делим его на 4, поскольку шаг
        // задается в байтах, а каждый пиксель занимает 4 байта.
        int index = i * lockedRect.Pitch / 4 + j;

        imageData[index] = 0xffff0000; // красный цвет
    }
}
_surface->UnlockRect();
```

Структура **D3DLOCKED_RECT** определена следующим образом:

```
typedef struct _D3DLOCKED_RECT {
    INT Pitch;   // шаг поверхности
    void *pBits; // указатель на начало памяти поверхности
} D3DLOCKED_RECT;
```

Приведем несколько комментариев к коду блокировки поверхности. Очень важно предположение о 32-разрядном формате пикселей, поскольку мы выполняем приведение к типу **DWORD**, который является 32-разрядным. Благодаря этому мы можем считать, что каждое значение **DWORD** представляет один пиксель. И не заморачивайтесь о том, как **0xffff0000** представляет красный цвет — мы поговорим об этом в главе 4.

1.3.2 Множественная выборка

Множественная выборка (*multisampling*) — это техника, использующаяся для сглаживания ступенчатых линий на изображении, представленном в виде матрицы пикселей. Наиболее частое применение множественной выборки — полноэкранное сглаживание (full-screen antialiasing) (рис. 1.3).

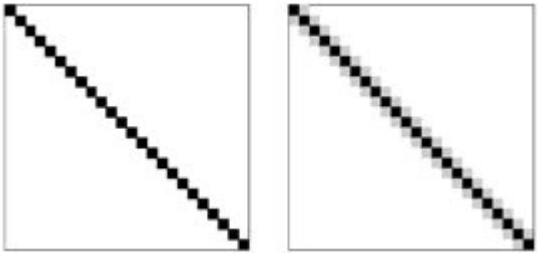

Рис. 1.3. *Слева изображена линия с зазубренным краем. Справа — та же линия, но с использованием множественной выборки, выглядящая более гладкой*

Перечисление **D3DMULTISAMPLE_TYPE** содержит константы, позволяющие задать уровень множественной выборки для поверхности.

- **D3DMULTISAMPLE_NONE** — Множественная выборка не используется.
- **D3DMULTISAMPLE_1_SAMPLE**...**D3DMULTISAMPLE_16_SAMPLE** — Устанавливает уровень множественной выборки от 1 до 16.

От типа множественной выборки зависит уровень качества. Тип констант — **DWORD**.

В примерах из этой книги множественная выборка не используется, поскольку она сильно замедляет работу приложений. Если вы захотите добавить ее, не забудьте вызвать метод **IDrect3D9::Check-DeviceMultiSampleType**, чтобы проверить поддерживает ли используемое устройство множественную выборку и определить допустимые уровни качества.

1.3.3 Формат пикселей

При создании поверхностей или текстур нам часто надо будет определить формат пикселей ресурсов Direct3D. Формат определяется как член перечисления **D3DFORMAT**. Вот некоторые форматы:

- **D3DFMT_R8G8B8** — 24-разрядный формат пикселей, где, начиная с самого левого разряда, 8 бит отведены для красного цвета, 8 бит — для зеленого и 8 бит — для синего.

- **D3DFMT_X8R8G8B8** — 32-разрядный формат пикселей, где, начиная с самого левого разряда, 8 бит не используются, 8 бит отведены для красного цвета, 8 бит — для зеленого и 8 бит — для синего.
- **D3DFMT_A8R8G8B8** — 32-разрядный формат пикселей, где, начиная с самого левого разряда, 8 бит используются для альфа-канала, 8 бит отведены для красного цвета, 8 бит — для зеленого и 8 бит — для синего.
- **D3DFMT_A16B16G16R16F** — 64-разрядный формат пикселей с плавающей запятой. Начиная с самого левого разряда, 16 бит используются для альфа-канала, 16 бит отведены для синего цвета, 16 бит — для зеленого и 16 бит — для красного.
- **D3DFMT_A32B32G32R32F** — 128-разрядный формат пикселей с плавающей запятой. Начиная с самого левого разряда, 32 разряда используются для альфа-канала, 32 разряда отведены для синего цвета, 32 разряда — для зеленого и 32 разряда — для красного.

Полный список поддерживаемых форматов пикселей приведен в описании перечисления **D3DFORMAT** в документации к SDK.

ПРИМЕЧАНИЕ Первые три формата (**D3DFMT_R8G8B8**, **D3DFMT_X8R8G8B8** и **D3DFMT_A8R8G8B8**) широко распространены и поддерживаются большинством современных видеокарт. Поддержка форматов с плавающей точкой и некоторых других форматов (упомянутых в документации SDK) встречается реже. При использовании малораспространенного формата не забудьте проверить перед его использованием, что установленная видеокарта поддерживает его.

1.3.4 Пул памяти

Поверхности и другие ресурсы Direct3D могут быть размещены в различных областях памяти. Используемая область памяти задается с помощью одной из констант перечисления **D3DPOOL**. Доступны следующие варианты:

- **D3DPOOL_DEFAULT** — Выбор используемого по умолчанию пула памяти позволяет Direct3D размещать ресурс в той области памяти, которая наиболее подходит для ресурсов данного типа с учетом его использования. Это может быть видеопамять, память AGP или системная память. Обратите внимание, что ресурсы, размещаемые в пуле по умолчанию, должны быть уничтожены (освобождены) до вызова метода **IDirect3DDevice9::Reset**, и могут быть повторно инициализированы после сброса.
- **D3DPOOL_MANAGED** — ресурсы размещаются в пуле памяти, управляемом Direct3D (это значит, что при необходимости они могут автоматически перемещаться устройством в видеопамять или память AGP). Кроме того, в системной памяти хранится резервная копия ресурса. Если приложение получает доступ или изменяет ресурс, работа ведется с

копией в системной памяти. Затем, если надо, Direct3D автоматически обновляет данные в видеопамяти.

- **D3DPOOL_SYSTEMMEM** — Ресурс будет размещен в системной памяти.
- **D3DPOOL_SCRATCH** — Указывает, что ресурс будет размещен в системной памяти. Отличие этого пула от **D3DPOOL_SYSTEMMEM** в том, что ресурс не должен соответствовать налагаемым графическим устройствам ограничениям. Следовательно, у графического устройства нет доступа к такому ресурсу. Тем не менее, эти ресурсы могут применяться в операциях копирования как в качестве источника, так и в качестве приемника.

1.3.5 Цепочка обмена и переключение страниц

Direct3D управляет набором поверхностей, обычно двумя или тремя, который называется *цепочка обмена* (*swap chain*) и представляется интерфейсом **IDirect3DSwapChain9**. Мы не будем подробно говорить об этом интерфейсе, поскольку всю работу берет на себя Direct3D и вмешиваться в его действия приходится очень редко. Вместо этого мы поговорим о его назначении и общих принципах работы.

Цепочка обмена и, говоря более точно, техника переключения страниц, используется для достижения плавности анимации. На рис. 1.4 показана цепочка обмена, состоящая из двух поверхностей.

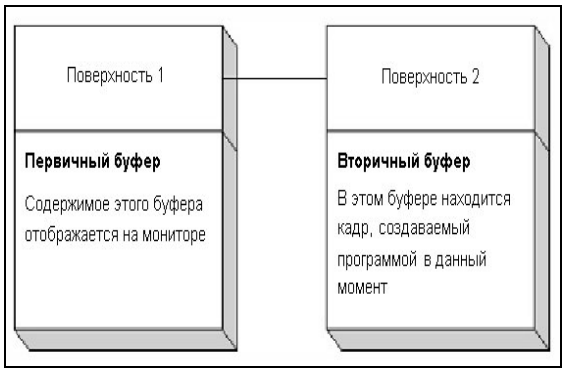

Рис. 1.4. *Цепочка обмена из двух поверхностей (первичного и вторичного буфера)*

На рис. 1.4 в *первичном буфере* (*front buffer*) находится та поверхность, которая в данный момент отображается на экране монитора. Монитор не показывает изображение находящееся в первичном буфере мгновенно; например, при установленной для монитора частоте кадров 60 Гц, вывод изображения занимает одну шестидесятую секунды. Частота кадров в приложении часто отличается от частоты кадров монитора (например, приложение может визуализировать кадр быстрее, чем монитор отобразит его). Мы не хотим обновлять содержимое первичного буфера, занося в него следующий кадр, в то время, когда монитор еще не закончил отображать текущий, но мы также не хотим останавливать

визуализацию кадров, чтобы подождать, пока монитор полностью выведет текущий кадр. Поэтому мы визуализируем кадр во внеэкранной поверхности (*вторичном буфере*, *back buffer*); затем, когда монитор закончит отображение поверхности из первичного буфера, мы перемещаем этот буфер в конец цепочки обмена, и передвигаем в цепочке вторичный буфер, чтобы он стал первичным.

Этот процесс называется *показом* (*presenting*). На рис. 1.5 показана цепочка обмена до и после показа.

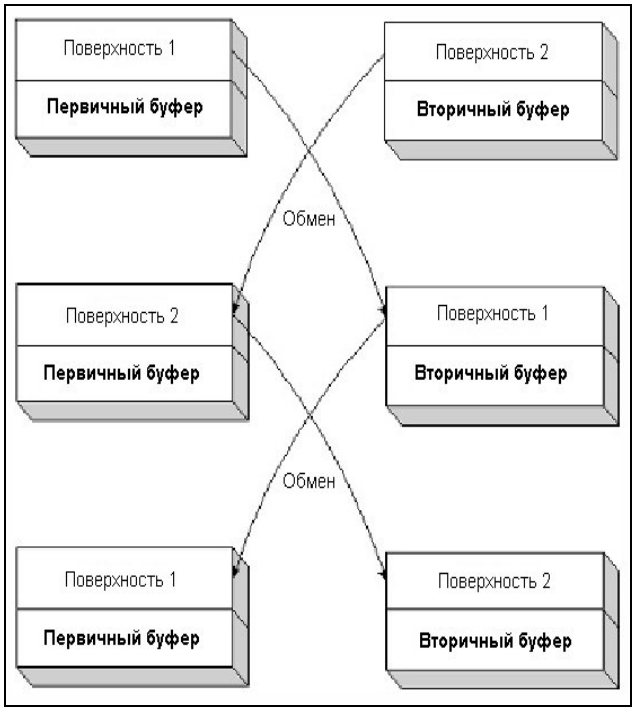

Рис. 1.5. *Показ. Используя содержащую две поверхности цепочку обмена, мы видим, что основу показа составляет переключение поверхностей*

Таким образом, структура кода визуализации будет следующей:

1. Визуализировать вторичный буфер.
2. Показать вторичный буфер.
3. Перейти к пункту 1.

1.3.6 Буфер глубины

Буфер глубины (*depth buffer*) — это поверхность, которая содержит не изображение, а информацию о глубине отдельных пикселей. Каждому пикселю в окончательном изображении соответствует элемент буфера глубины. Так, если размер окончательного изображения 640 × 480 пикселей, в буфере глубины должно быть 640 × 480 элементов.

На рис. 1.6 показана простая сцена, где одни объекты частично скрывают другие, находящиеся позади них. Чтобы определить, пиксели какого объекта

находятся поверх всех других, Direct3D использует технологию, называемую *буфером глубины* (*depth buffer*) или *z-буферизацией* (*z-buffering*).

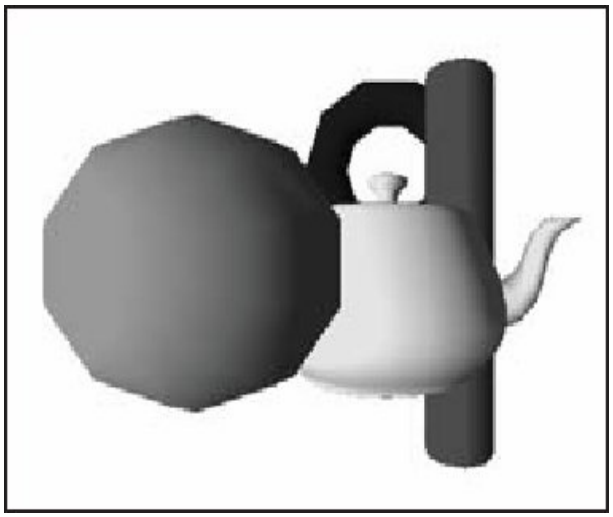

Рис. 1.6. *Группа объектов, частично закрывающих друг друга*

Буфер глубины работает путем вычисления значения глубины каждого пикселя и последующего выполнения проверки глубины. Проверка глубины заключается в сравнении значений глубины пикселов, расположенных в заданной позиции. Сравнение выигрывает тот пиксель, который находится ближе всего к камере, и именно он будет сохранен. Это имеет смысл, поскольку самый близкий к камере пиксель скрывает все остальные пиксели, находящиеся позади него.

Формат буфера глубины определяет точность сравнения глубины пикселей. То есть 24-разрядный буфер глубины обеспечивает более высокую точность, чем 16-разрядный. Большинство приложений замечательно работают с 24-разрядным буфером глубины, хотя Direct3D поддерживает и 32-разрядный буфер.

- **`D3DFMT_D32`** — 32-разрядный буфер глубины.
- **`D3DFMT_D24S8`** — 24-разрядный буфер глубины с 8 разрядами, зарезервированными для буфера трафарета.
- **`D3DFMT_D24X8`** — 24-разрядный буфер глубины.
- **`D3DFMT_D24X4S4`** — 24-разрядный буфер глубины с 4 разрядами, зарезервированными под буфер трафарета.
- **`D3DFMT_D16`** — 16-разрядный буфер глубины.

ПРИМЕЧАНИЕ Буфер трафарета представляет собой более сложную тему, которая будет подробно разобрана в главе 8.

1.3.7 Обработка вершин

Вершины представляют собой основные строительные блоки трехмерной геометрии и могут обрабатываться двумя различными способами: программно (*программная обработка вершин*) или аппаратурой видеокарты (*аппаратная*

обработка вершин). Программная обработка вершин всегда поддерживается и может быть использована. Аппаратная обработка вершин, с другой стороны, может использоваться только если она поддерживается установленной видеокартой.

Аппаратная обработка вершин всегда предпочтительнее, поскольку выполняется гораздо быстрее, чем программная. Кроме того, аппаратная обработка вершин освобождает центральный процессор, который, благодаря этому, может выполнять другие задачи.

ПРИМЕЧАНИЕ Иногда, вместо того, чтобы сказать, что видеокарта поддерживает аппаратную обработку вершин, говорят что видеокарта поддерживает аппаратную обработку преобразований и расчета освещения.

1.3.8 Возможности устройств

Каждой возможности, предлагаемой Direct3D соответствует член данных или бит в структуре **D3DCAPS9**. Идея состоит в том, чтобы инициализировать экземпляр структуры **D3DCAPS9** на основании фактических возможностей конкретного устройства. Затем, в нашем приложении, мы можем проверить, поддерживает ли устройство требуемую возможность, проверив соответствующий член данных или бит в экземпляре **D3DCAPS9**.

Это иллюстрирует следующий пример. Предположим, мы хотим проверить способно ли устройство осуществлять аппаратную обработку вершин (или, другими словами, поддерживает ли устройство аппаратную обработку преобразований и расчета освещения). Посмотрев описание структуры **D3DCAPS9** в документации SDK, мы обнаружим, что бит **D3DDEVCAPS_HWTRANSFORMANDLIGHT** в члене данных **D3DCAPS9::DevCaps** указывает, поддерживает ли устройство аппаратную обработку преобразований и расчета освещенности. Следовательно наша проверка, подразумевая, что **caps** — это экземпляр структуры **D3DCAPS9**, будет выглядеть следующим образом:

```
bool supportsHardwareVertexProcessing;

// Если бит установлен, значит данная возможность
// поддерживается устройством
if(caps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT)
{
    // Бит установлен - возможность поддерживается
    supportsHardwareVertexProcessing = true;
}
else
{
    // Бит сброшен - возможность не поддерживается
    hardwareSupportsVertexProcessing = false;
}
```

ПРИМЕЧАНИЕ	DevCaps означает «возможности устройства» (device capabilities).

ПРИМЕЧАНИЕ	Как инициализировать экземпляр структуры **D3DCAPS9** на основании возможностей конкретного устройства мы покажем в следующем разделе.

ПРИМЕЧАНИЕ	Мы рекомендуем вам посмотреть описание структуры **D3DCAPS9** в документации к SDK и изучить полный список возможностей устройств, поддерживаемых в Direct3D.

1.4 Инициализация Direct3D

В следующих подразделах будет показано как инициализировать Direct3D. Процесс инициализации Direct3D может быть разбит на следующие этапы:

1. Запрос указателя на интерфейс **IDirect3D9**. Этот интерфейс применяется для получения информации об установленных в компьютере устройствах и создания интерфейса **IDirect3DDevice9**, являющимся нашим объектом C++, представляющим аппаратные устройства, используемые для вывода трехмерной графики.
2. Проверка возможностей устройства (**D3DCAPS9**), чтобы узнать поддерживает ли первичный видеоадаптер (основная видеокарта) аппаратную обработку вершин или нет. Мы должны знать это, чтобы создать интерфейс **IDirect3DDevice9**.
3. Инициализация экземпляра структуры **D3DPRESENT_PARAMETERS**. Эта структура содержит ряд параметров, позволяющих нам задать характеристики интерфейса **IDirect3DDevice9**, который мы намереваемся создать.
4. Создание объекта **IDirect3DDevice9**, основываясь на инициализированной структуре **D3DPRESENT_PARAMETERS**. Как говорилось ранее, объект **IDirect3DDevice9** — это наш объект C++, представляющий аппаратные устройства, используемые для отображения трехмерной графики.

Помните, что в этой книге для отображения трехмерной графики мы используем первичный адаптер. Если в вашей системе только одна видеокарта, она и будет первичным адаптером. Если у вас несколько видеокарт, то первичной будет та, которую вы используете в данный момент (т.е. та, которая отображает рабочий стол Windows и т.д.).

1.4.1 Запрос интерфейса IDirect3D9

Инициализация Direct3D начинается с запроса указателя на интерфейс **IDirect3D9**. Это выполняется с помощью простого вызова специальной функции Direct3D, как показано в приведенном ниже фрагменте кода:

```
IDirect3D9* _d3d9;
_d3d9 = Direct3DCreate9(D3D_SDK_VERSION);
```

Единственным параметром функции **Direct3DCreate9** всегда должна быть константа **D3D_SDK_VERSION**, гарантирующая, что при построении приложения будут использованы правильные заголовочные файлы. Если при работе функции возникли ошибки, она возвращает нулевой указатель.

Объект **IDirect3D9** используется для двух вещей: перечисления устройств и создания объекта **IDirect3DDevice9**. Перечисление устройств подразумевает определение возможностей, видеорежимов, форматов и другой информации о каждой установленной в системе видеокарте. Например, чтобы создать представляющий физическое устройство объект **IDirect3DDevice9**, необходимо указать поддерживаемую устройством конфигурацию видеорежима и формата. Чтобы найти такую работающую конфигурацию, используются методы перечисления **IDirect3D9**.

Однако, поскольку перечисление устройств может быть достаточно сложной задачей, а мы хотим как можно быстрее перейти к работе с Direct3D, мы решили *не выполнять* перечисление устройств, за исключением единственной проверки, о которой будет рассказано в следующем разделе. Чтобы отсутствие перечисления не привело к проблемам, мы выбрали «безопасную» конфигурацию, которую должны поддерживать почти все аппаратные устройства.

1.4.2 Проверка поддержки аппаратной обработки вершин

Создавая объект **IDirect3DDevice9**, представляющий первичный видеоадаптер, мы должны указать используемый им способ обработки вершин. Мы хотим по-возможности использовать аппаратную обработку вершин, но не все видеокарты поддерживают ее и мы сперва должны проверить наличие поддержки аппаратной обработки вершин.

Чтобы сделать это мы сперва должны инициализировать экземпляр **D3DCAPS9** на основании возможностей первичного видеоадаптера. Воспользуемся следующим методом:

```
HRESULT IDirect3D9::GetDeviceCaps(
     UINT Adapter,
     D3DDEVTYPE DeviceType,
     D3DCAPS9 *pCaps
);
```

- **Adapter** — указывает физический видеоадаптер, возможности которого мы хотим определить.
- **DeviceType** — задает тип устройства (т.е. аппаратное устройство (**D3DDEVTYPE_HAL**) или программное устройство (**D3DDEVTYPE_REF**)).

- **pCaps** — возвращает инициализированную структуру с описанием возможностей устройства.

Теперь мы можем проверить возможности устройства, как описывалось в разделе 1.3.8. Это иллюстрирует следующий фрагмент кода:

```
// Заполняем структуру D3DCAPS9 информацией о
// возможностях первичного видеоадаптера.

D3DCAPS9 caps;
d3d9->GetDeviceCaps(
      D3DADAPTER_DEFAULT, // Означает первичный видеоадаптер
      deviceType,   // Задает тип устройства, обычно D3DDEVTYPE_HAL
      &caps);       // Возвращает заполненную структуру D3DCAPS9,
                    // которая содержит информацию о возможностях
                    // первичного видеоадаптера.

// Поддерживается аппаратная обработка вершин?
int vp = 0;
if(caps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT)
{
    // да, сохраняем в vp флаг поддержки аппаратной
    // обработки вершин.
    vp = D3DCREATE_HARDWARE_VERTEXPROCESSING;
}
else
{
    // Нет, сохраняем в vp флаг использования программной
    // обработки вершин.
    vp = D3DCREATE_SOFTWARE_VERTEXPROCESSING;
}
```

Обратите внимание, что мы сохраняем способ обработки вершин, который будем использовать, в переменной **vp**. Это вызвано тем, что нам надо будет указать используемый тип обработки вершин, когда позднее мы будем создавать объект **IDirect3DDevice9**.

ПРИМЕЧАНИЕ Идентификаторы **D3DCREATE_HARDWARE_VERTEXPROCESSING** и **D3DCREATE_SOFTWARE_VERTEXPROCESSING** — это предопределенные константы, означающие соответственно аппаратную обработку вершин и программную обработку вершин.

СОВЕТ Разрабатывая приложения, использующие новые, специальные или дополнительные возможности (другими словами, возможности не поддерживаемые большинством устройтв) следует всегда проверять возможности устройства (**D3DCAPS9**), чтобы перед использованием требуемой возможности убедиться, что устройство поддерживает ее. Никогда не предполагайте, что необходимая вам возможность будет доступна. Также обраите внимание, что приложения из этой книги в большинстве случаев не следуют данной рекомендации — в большинстве случаев мы не проверяем возможности устройств.

ПРИМЕЧАНИЕ Если какое-либо из рассматриваемых в книге приложений не работает на вашей системе, это скорее всего вызвано тем, что ваша аппаратура не поддерживает возможности, необходимые приложению. В таких случаях попробуйте использовать устройство REF.

1.4.3 Заполнение структуры D3DPRESENT_PARAMETERS

Следующий этап процесса инициализации — заполнение экземпляра структуры **D3DPRESENT_PARAMETERS**. Эта структура используется для задания ряда характеристик объекта **IDirect3DDevice9**, который мы будем создавать, и объявлена следующим образом:

```
typedef struct _D3DPRESENT_PARAMETERS_ {
    UINT      BackBufferWidth;
    UINT      BackBufferHeight;
    D3DFORMAT BackBufferFormat;
    UINT      BackBufferCount;
    D3DMULTISAMPLE_TYPE MultiSampleType;
    DWORD     MultiSampleQuality;
    D3DSWAPEFFECT SwapEffect;
    HWND      hDeviceWindow;
    BOOL      Windowed;
    BOOL      EnableAutoDepthStencil;
    D3DFORMAT AutoDepthStencilFormat;
    DWORD     Flags;
    UINT      FullScreen_RefreshRateInHz;
    UINT      PresentationInterval;
} D3DPRESENT_PARAMETERS;
```

ПРИМЕЧАНИЕ В приведенном ниже описании членов структуры **D3DPRESENT_PARAMETERS** перечислены только те из них, которые, по нашему мнению, наиболее важны для новичков. За полным описанием структуры мы отсылаем вас к документации SDK.

- **BackBufferWidth** — Ширина поверхности вторичного буфера в пикселах.
- **BackBufferHeight** — Высота поверхности вторичного буфера в пикселах.
- **BackBufferFormat** — Формат пикселей во вторичном буфере (т.е. для 32-разрядного формата укажите **D3DFMT_A8R8G8B8**).
- **BackBufferCount** — Количество используемых вторичных буферов. Обычно мы задаем значение 1, чтобы указать, что нам нужен только один вторичный буфер.

- **MultiSampleType** — Используемый для вторичного буфера тип множественной выборки. Подробная информация об этом параметре содержится в документации SDK.
- **MultiSampleQuality** — Уровень качества множественной выборки. Подробная информация об этом параметре содержится в документации SDK.
- **SwapEffect** — Член перечисления **D3DSWAPEFFECT**, указывающий, как будет осуществляться переключение буферов в цепочке переключений. Для большинства случаев можно указать значение **D3DSWAPEFFECT_DISCARD**.
- **hDeviceWindow** — Дескриптор связанного с устройством окна. Укажите дескриптор того окна приложения, в котором вы хотите выводить изображение.
- **Windowed** — Укажите **true** для запуска приложения в оконном режиме или **false** — для работы в полноэкранном режиме.
- **EnableAutoDepthStencil** — укажите значение **true**, чтобы Direct3D автоматически создал и поддерживал буферы глубины и трафарета.
- **AutoDepthStencilFormat** — формат буферов глубины и трафарета (например, для 24-разрядного буфера глубины с 8 разрядами, зарезервированными для буфера трафарета, укажите **D3DFMT_D24S8**).
- **Flags** — Дополнительные параметры. Укажите ноль (если флаги отсутствуют) или член набора **D3DPRESENTFLAG**. Полный список допустимых флагов приведен в документации. Здесь мы рассмотрим два наиболее часто используемых:
  - **D3DPRESENTFLAG_LOCKABLE_BACKBUFFER** — Указывает, что вторичный буфер может быть заблокирован. Обратите внимание, что использование блокируемого вторичного буфера снижает производительность.
  - **D3DPRESENTFLAG_DISCARD_DEPTHSTENCIL** — указывает, что буфер глубины и трафарета после показа вторичного буфера становится некорректным. Под словом «некорректным» мы подразумеваем, что данные, хранящиеся в буфере глубины и трафарета могут стать неверными. Это может увеличить производительность.
- **FullScreen_RefreshRateInHz** — Частота кадров; используйте частоту кадров по умолчанию, указав значение **D3DPRESENT_RATE_DEFAULT**.
- **PresentationInterval** — Член набора **D3DPRESENT**. Полный список допустимых значений приведен в документации. Наиболее часто используются следующие два:

- **D3DPRESENT_INTERVAL_IMMEDIATE** — Показ выполняется немедленно.
- **D3DPRESENT_INTERVAL_DEFAULT** — Direct3D сам выбирает частоту показа. Обычно она равна частоте кадров.

Вот пример заполнения структуры:

```
D3DPRESENT_PARAMETERS d3dpp;

d3dpp.BackBufferWidth            = 800;
d3dpp.BackBufferHeight           = 600;
d3dpp.BackBufferFormat           = D3DFMT_A8R8G8B8; //формат пикселей
d3dpp.BackBufferCount            = 1;
d3dpp.MultiSampleType            = D3DMULTISAMPLE_NONE;
d3dpp.MultiSampleQuality         = 0;
d3dpp.SwapEffect                 = D3DSWAPEFFECT_DISCARD;
d3dpp.hDeviceWindow              = hwnd;
d3dpp.Windowed                   = false; // полноэкранный режим
d3dpp.EnableAutoDepthStencil     = true;
d3dpp.AutoDepthStencilFormat     = D3DFMT_D24S8; // формат буфера
                                                 // глубины
d3dpp.Flags                      = 0;
d3dpp.FullScreen_RefreshRateInHz = D3DPRESENT_RATE_DEFAULT;
d3dpp.PresentationInterval       = D3DPRESENT_INTERVAL_IMMEDIATE;
```

1.4.4 Создание интерфейса IDirect3DDevice9

Заполнив структуру **D3DPRESENT_PARAMETERS** мы можем создать объект **IDirect3DDevice9** с помощью следующего метода:

```
HRESULT IDirect3D9::CreateDevice(
     UINT Adapter,
     D3DDEVTYPE DeviceType,
     HWND hFocusWindow,
     DWORD BehaviorFlags,
     D3DPRESENT_PARAMETERS *pPresentationParameters,
     IDirect3DDevice9** ppReturnedDeviceInterface
);
```

- **Adapter** — указывает физический видеоадаптер, который будет представлять создаваемый объект **IDirect3DDevice9**.
- **DeviceType** — задает тип используемого устройства (т.е. аппаратное устройство (**D3DDEVTYPE_HAL**) или программное устройство (**D3DDEVTYPE_REF**)).
- **hFocusWindow** — дескриптор окна с которым будет связано устройство. Обычно это то окно, в которое будет выводиться изображение и для наших целей мы здесь задаем тот же дескриптор, который указан в члене **d3dpp.hDeviceWindow** структуры **D3DPRESENT_PARAMETERS**.

- **BehaviorFlags** — в этом параметре указывается значение **D3DCREATE_HARDWARE_VERTEXPROCESSING** либо **D3DCREATE_SOFTWARE_VERTEXPROCESSING**.
- **pPresentationParameters** — указывается инициализированный экземпляр структуры **D3DPRESENT_PARAMETERS**, задающий параметры устройства.
- **ppReturnedDeviceInterface** — возвращает указатель на созданное устройство.

Вот пример использования функции:

```
IDirect3DDevice9* device = 0;
hr = d3d9->CreateDevice(
       D3DADAPTER_DEFAULT, // первичный видеоадаптер
       D3DDEVTYPE_HAL,     // тип устройства
       hwnd,               // окно, связанное с устройством
       D3DCREATE_HARDWARE_VERTEXPROCESSING, // тип обработки вершин
       &d3dpp,             // параметры показа
       &device);           // возвращает созданное устройство

if(FAILED(hr))
{
    ::MessageBox(0, "CreateDevice() - FAILED", 0, 0);
    return 0;
}
```

1.5 Пример приложения: инициализация Direct3D

В рассматриваемом в этой главе примере мы выполним инициализацию Direct3D-приложения и очистим экран, заполнив его черным цветом (рис. 1.7).

Рис. 1.7. *Окно программы, рассматриваемой в этой главе*

Этот пример, как и все другие примеры в этой книге, использует код из файлов d3dUtility.h и d3dUtility.cpp, которые можно скачать с веб-сайта, посвященного этой книге. Эти файлы содержат функции, реализующие общие задачи, которые должно выполнять каждое Direct3D-приложение, такие как создание окна, инициализация Direct3D и вход в цикл обработки сообщений. Благодаря созданию функций-оберток для этих задач, в коде примеров мы можем сосредоточиться на рассматриваемой в соответствующей главе теме. Кроме того, на протяжении книги мы будем добавлять к этим файлам полезный вспомогательный код.

1.5.1 Файлы d3dUtility.h/cpp

Перед тем, как перейти к примеру из этой главы, давайте потратим немного времени, чтобы познакомиться с функциями, предоставляемыми файлами d3dUtility.h/cpp. Вот как выглядит код из файла d3dUtility.h:

```
// Включение основного заголовочного файла Direct3DX. В нем
// осуществляется включение других, необходимых нам заголовочных
// файлов Direct3D.
#include <d3dx9.h>

namespace d3d
{
    bool InitD3D(
         HINSTANCE hInstance,       // [in] Экземпляр приложения.
         int width, int height,     // [in] Размеры вторичного буфера.
         bool windowed,             // [in] Оконный (true) или
                                    //    полноэкранный (false) режим.
         D3DDEVTYPE deviceType,     // [in] HAL или REF
         IDirect3DDevice9** device); // [out] Созданное устройство.

    int EnterMsgLoop(
         bool (*ptr_display)(float timeDelta));

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

    template<class T> void Release(T t)
    {
        if(t) {
            t->Release();
            t = 0;
        }
    }
    template<class T> void Delete(T t)
    {
        if(t) {
            delete t;
            t = 0;
        }
    }
}
```

- **InitD3D** — Эта функция инициализирует главное окно приложения и содержит код инициализации Direct3D, который обсуждался в разделе 1.4. Если функция завершается нормально, она возвращает указатель на созданный интерфейс **IDirect3DDevice9**. Обратите внимание, что параметры функции позволяют задать размеры окна и то, в каком режиме — оконном или полноэкранном — будет работать приложение. Чтобы познакомиться с деталями реализации, посмотрите код примера.
- **EnterMsgLoop** — Эта функция является оберткой для цикла обработки сообщений приложения. Она получает указатель на *функцию визуализации*. Функция визуализации — это функция в которой находится код для вывода создаваемого в примере изображения. Циклу сообщений необходимо знать, какая функция используется для визуализации, чтобы он мог вызывать ее и отображать сцену во время ожидания сообщений.

```
int d3d::EnterMsgLoop(bool(*ptr_display)(float timeDelta))
{
    MSG msg;
    ::ZeroMemory(&msg, sizeof(MSG));

    static float lastTime = (float)timeGetTime();
    while(msg.message != WM_QUIT)
    {
        if(::PeekMessage(&msg, 0, 0, 0, PM_REMOVE))
        {
            ::TranslateMessage(&msg);
            ::DispatchMessage(&msg);
        }
        else
        {
            float currTime = (float)timeGetTime();
            float timeDelta = (currTime - lastTime) * 0.001f;
            ptr_display(timeDelta); // вызов функции
                                    // визуализации
            lastTime = currTime;
        }
    }
return msg.wParam;
}
```

Часть кода занимается вычислением времени, прошедшего между обращениями к функции **ptr_display**, то есть времени между кадрами.

- **Release** — Этот шаблон разработан в качестве вспомогательной функции для освобождения COM-интерфейсов и присваивания указателям на них нулевых значений.
- **Delete** — Этот шаблон разработан в качестве вспомогательной функции для удаления объектов, освобождения занимаемой ими памяти и присваивания указателям на них нулевых значений.
- **WndProc** — Объявление оконной процедуры для главного окна приложения.

1.5.2 Каркас примера

Под *каркасом примера* мы подразумеваем общую структуру кода, которой придерживаются все рассматриваемые в этой книге примеры программ. Для каждого приложения мы в обязательном порядке будем реализовать три функции, не считая процедуры обработки сообщений и функции **`WinMain`**. В этих трех функциях будет реализоваться код, специфичный для конкретного приложения. Вот эти функции:

- **`bool Setup()`** — Это функция в которой инициализируется все, что должно быть инициализировано для данного приложения. В ней осуществляется выделение ресурсов, проверка возможностей устройств и установка состояний приложения.
- **`void Cleanup()`** — В этой функции мы освобождаем все ресурсы, выделенные для приложения в функции **`Setup`**, в основном это освобождение памяти.
- **`bool Display(float timeDelta)`** — В эту функцию мы помещаем весь код, отвечающий за рисование и код, который должен выполняться при переходе от кадра к кадру, например выполняющий изменение местоположения объектов. Параметр **`timeDelta`** — это время, прощедшее с момента вывода предыдущего кадра и используется он для синхронизации анимации с частотой смены кадров.

1.5.3 Приложение D3D Init

Как было сказано, рассматриваемый пример приложения создает и инициализирует Direct3D-приложение и очищает экран, заполняя его черным цветом. Обратите внимание, что для упрощения инициализации мы используем наши вспомогательные функции. Полный код проекта можно скачать с веб-сайта этой книги.

ПРИМЕЧАНИЕ	Рассматриваемый пример иллюстрирует те же идеи, что и урок Tutorial 1 из документации DirectX SDK. Завершив чтение этой главы, вы можете прочесть урок Tutorial 1, чтобы познакомиться с другой точкой зрения.

Мы начинаем с включения заголовочного файла d3dUtility.h и объявления глобальной переменной для устройства:

```
#include "d3dUtility.h"
IDirect3DDevice9* Device = 0;
```

Затем мы реализуем функции, входящие в каркас приложения:

```
bool Setup()
{
    return true;
}
```

```
void Cleanup()
{

}
```

В данном примере нам не требуются никакие ресурсы, так что методы **Setup** и **Cleanup** остаются пустыми.

```
bool Display(float timeDelta)
{
    if(Device)
    {
        Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                      0x00000000, 1.0f, 0);
        Device->Present(0, 0, 0, 0); // показ вторичного буфера
    }
    return true;
}
```

Метод **Display** вызывает метод **IDirect3DDevice9::Clear**, который очищает вторичный буфер и буфер глубины/трафарета, заполняя их черным цветом и константой 1.0 соответственно. Обратите внимание, что если приложение не остановлено, мы выполняем только код рисования. Объявление функции **IDirect3DDevice9::Clear** выглядит так:

```
HRESULT IDirect3DDevice9::Clear(
     DWORD Count,
     const D3DRECT* pRects,
     DWORD Flags,
     D3DCOLOR Color,
     float Z,
     DWORD Stencil
);
```

- **Count** — Количество прямоугольников в массиве **pRects**.
- **pRects** — Массив очищаемых прямоугольных областей экрана. Он позволяет очищать отдельные фрагменты поверхности.
- **Flags** — Указывает, какую поверхность очищать. Можно указывать одну или несколько из следующих поверхностей:
  - **D3DCLEAR_TARGET** — Целевая поверхность визуализации, обычно это вторичный буфер.
  - **D3DCLEAR_ZBUFFER** — Буфер глубины.
  - **D3DCLEAR_STENCIL** — Буфер трафарета.
- **Color** — Цвет, которым будет заполнена поверхность визуализации.
- **Z** — Значение, которым будет заполнен буфер глубины (z-буфер).
- **Stencil** — Значение, которым будет заполнен буфер трафарета.

После того, как поверхность очищена, мы показываем вторичный буфер, вызвав метод **IDirect3DDevice9::Present**.

Оконная процедура обрабатывает пару событий, а именно позволяет выходить из приложения, нажав клавишу **Esc**.

```
LRESULT CALLBACK d3d::WndProc(HWND hwnd, UINT msg, WPARAM wParam,
                              LPARAM lParam)
{
    switch( msg )
    {
    case WM_DESTROY:
         ::PostQuitMessage(0);
         break;

    case WM_KEYDOWN:
         if( wParam == VK_ESCAPE )
             ::DestroyWindow(hwnd);
         break;
    }
    return ::DefWindowProc(hwnd, msg, wParam, lParam);
}
```

Функция **WinMain** выполняет следующие действия:

1. Инициализирует главное окно и Direct3D.
2. Вызывает процедуру **Setup** для инициализации приложения.
3. Запускает цикл обработки сообщений, указав в качестве функции визуализации функцию **Display**.
4. Освобождает выделенные приложению ресурсы и объект **IDirect3DDevice9**.

```
int WINAPI WinMain(HINSTANCE hinstance,
                   HINSTANCE prevInstance,
                   LPSTR cmdLine,
                   int showCmd)
{
    if(!d3d::InitD3D(hinstance, 800, 600,
                  true, D3DDEVTYPE_HAL, &Device))
    {
        ::MessageBox(0, "InitD3D() - FAILED", 0, 0);
        return 0;
    }

    if(!Setup())
    {
        ::MessageBox(0, "Setup() - FAILED", 0, 0);
        return 0;
    }

    d3d::EnterMsgLoop( Display );
```

```
    Cleanup();

    Device->Release();

    return 0;
}
```

Как видите, благодаря вспомогательным функциям выполняющим обработку сообщений и инициализацию Direct3D, структура шаблона приложения получилась исключительно прозрачной.

Для большинства примеров из этой книги наша задача будет заключаться в написании реализаций функций **`Setup`**, **`Cleanup`** и **`Display`**.

ПРИМЕЧАНИЕ Помните, что для построения данного приложения компоновщику необходимы библиотеки d3d9.lib, d3dx9.lib и winmm.lib.

1.6 Итоги

- Direct3D можно представлять как посредника между программистом и графическим оборудованием. программист вызывает функции Direct3D, которые в свою очередь, приказывают графическому оборудованию выполнить необходимые операции, взаимодействуя с ним через уровень абстрагирования от оборудования (HAL) устройства.
- Устройство REF позволяет разработчикам тестировать те возможности, которые предлагаются Direct3D, но не реализованы в используемом оборудовании.
- Модель компонентных объектов (COM) — это технология, позволяющая DirectX быть независимым от языка программирования и совместимым со всеми предыдущими версиями. Работающим с Direct3D программистам не требуется детальное знание особенностей и принципов работы COM; достаточно знать как запросить и освободить COM-интерфейс.
- Поверхность — это специальный интерфейс Direct3D, используемый для хранения двухмерных изображений. Формат пикселей поверхности задается с помощью членов перечисления **`D3DFORMAT`**. Поверхности и другие ресурсы Direct3D могут хранится в нескольких различных пулах памяти, определяемых с помощью членов перечисления **`D3DPOOL`**. Кроме того, поверхности могут использовать множественную выборку, в результате чего края объектов будут выглядеть более гладкими.
- Интерфейс **`IDirect3D9`** применяется для получения информации об установленных в системе графических устройствах. Например, через этот интерфейс мы можем получить описание возможностей устройства. Кроме того, он используется для создания интерфейса **`IDirect3DDevice9`**.

- Интерфейс **`IDirect3DDevice9`** можно представлять себе как программное средство управления графическим устройством. Например, вызов метода **`IDirect3DDevice9::Clear`** прикажет устройству очистить указанную поверхность.
- Каркас примера используется для обеспечения единой согласованной структуры для всех рассматриваемых в книге приложений. Вспомогательный код из файлов d3dUtility.h/cpp является оберткой для кода инициализации, который должен быть реализован в каждом приложении. Создав эти обертки мы скрываем код, что позволяет в примерах сосредоточиться непосредственно на рассматриваемой теме.

Глава 2
Конвейер визуализации

Главной темой этой главы является конвейер визуализации (rendering pipeline). Конвейер визуализации отвечает за создание двухмерного изображения на основании геометрического описания трехмерного мира и виртуальной камеры, определяющей точку с которой зритель смотрит на этот мир.

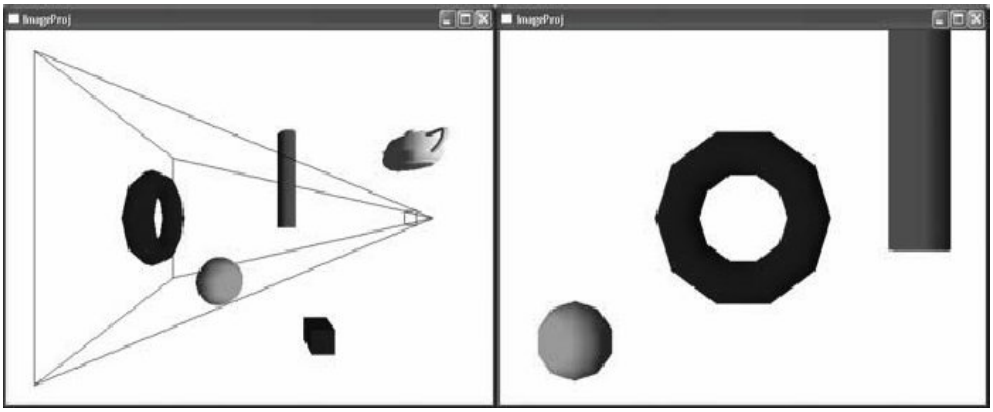

Рис. 2.1. *Левое изображение показывает несколько объектов, образующих трехмерную сцену и нацеленную на них камеру. Справа показано двухмерное изображение, созданное на основании того, что «видит» камера.*

Цели

- Узнать, как в Direct3D представляются трехмерные объекты.
- Изучить моделирование виртуальной камеры.
- Познакомиться с конвейером визуализации — процессом генерации двухмерного изображения на основании математического описания трехмерного мира.

2.1 Представление моделей

Сценой (*scene*) называется набор объектов или моделей. Объект представляется с помощью *сетки с треугольными ячейками* (*triangle mesh*), как показано на рис. 2.2. Отдельные треугольники сетки — это строительные блоки с помощью которых мы моделируем объекты. Чтобы сослаться на треугольник сетки мы будем использовать следующие взаимозаменяемые термины: полигон, примитив и ячейка сетки. (Треугольники являются примитивами, но Direct3D поддерживает еще два вида примитивов: линии и точки. Однако, поскольку линии и точки не слишком полезны для моделирования трехмерных твердых объектов, мы опустим обсуждение этих примитивов. О некоторых применениях точек мы поговорим в главе 14.)

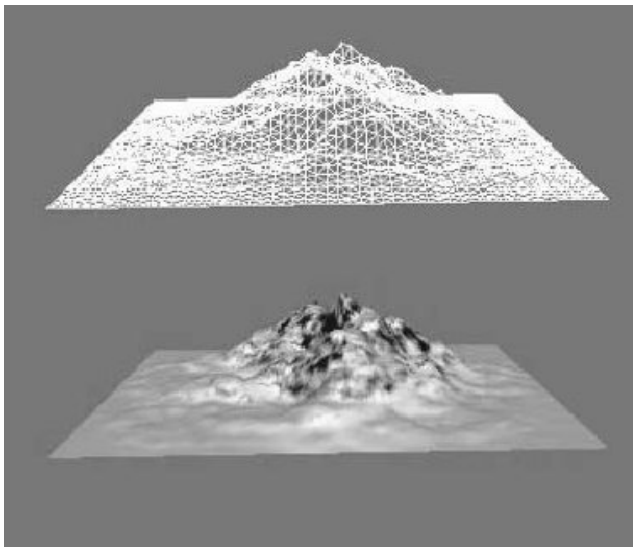

Рис. 2.2. *Ландшафт, представленный с помощью сетки из треугольников*

Точка, в которой встречаются две грани полигона называется *вершиной* (*vertex*). Чтобы описать треугольник, мы задаем местоположение трех точек, являющихся его вершинами (рис. 2.3.). Чтобы описать объект, мы задаем составляющие его треугольники.

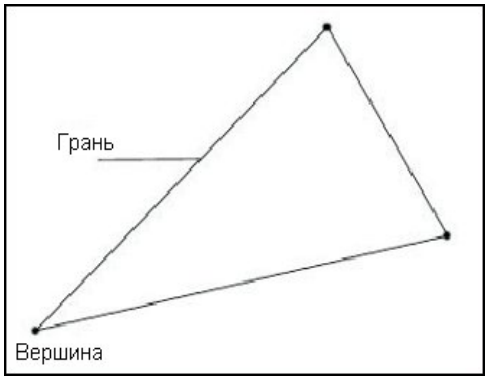

Рис. 2.3. *Треугольник, заданный тремя вершинами*

2.1.1 Форматы вершин

Приведенное выше определение вершин верно с математической точки зрения, но в контексте Direct3D является неполным. Это вызвано тем, что в Direct3D у вершины могут быть дополнительные свойства, помимо ее местоположения. Например, вершине может быть назначен цвет или с ней может быть связана нормаль (цвет будет обсуждаться в главе 4, а нормали — в главе 5). Direct3D обладает значительной гибкостью и позволяет нам конструировать собственные форматы вершин; другими словами, он позволяет нам указать, какая информация будет содержаться в данных вершины.

Чтобы создать собственный формат вершин нам сначала необходимо создать структуру, которая будет хранить необходимые нам данные вершины. Ниже для примера мы приводим два различных формата вершин: один хранит местоположение и цвет, а другой — местоположение, нормаль и координаты текстуры (о текстурах рассказывается в главе 6).

```
struct ColorVertex
{
    float _x, _y, _z; // местоположение
    DWORD _color;     // цвет
};

struct NormalTexVertex
{
    float _x, _y, _z;    // местоположение
    float _nx, _ny, _nz; // вектор нормали
    float _u, _v;        // координаты текстуры
};
```

После того, как мы завершили объявление структуры данных вершины, нам необходимо описать формат хранения этих данных в структуре с помощью комбинации флагов *настраиваемого формата вершин* (*flexible vertex format*, FVF). Для первой из представленных выше структур данных вершин мы получаем следующее описание формата:

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

Это описание говорит о том, что структура данных вершины, соответствующая данному формату вершин содержит сведения о местоположении и информацию о цвете.

```
#define FVF_NORMAL_TEX (D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1)
```

В этом описании говорится, что структура данных вершины, соответствующая данному формату, содержит данные о местоположении, нормали и координатах текстуры.

Вы должны помнить об одном ограничении — флаги настраиваемого формата вершин должны располагаться в том же самом порядке, что и соответствующие им поля в структуре данных вершины.

Полный список доступных флагов формата вершин вы найдете в документации по ключевому слову D3DFVF.

2.1.2 Треугольники

Треугольники являются основными строительными блоками трехмерных объектов. Чтобы сконструировать объект мы создаем список треугольников, описывающих его форму и контуры. Список треугольников содержит данные каждого треугольника, который мы будем отображать. Например, чтобы создать прямоугольник, мы разбиваем его на два треугольника, как показано на рис. 2.4, и задаем вершины каждого треугольника.

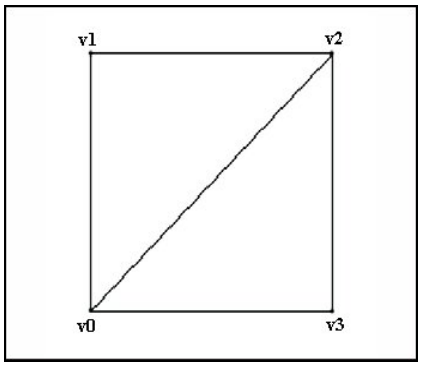

Рис. 2.4. *Прямоугольник, образованный из двух треугольников*

```
Vertex rect[6] = {v0, v1, v2,  // треугольник 0
                  v0, v2, v3}; // треугольник 1
```

ПРИМЕЧАНИЕ Порядок, в котором задаются вершины треугольника очень важен и называется *порядком обхода* (*winding order*). Подробнее об этом мы поговорим в разделе 2.3.4.

2.1.3 Индексы

Очень часто образующие трехмерный объект треугольники имеют общие вершины, как, например в прямоугольнике, изображенном на рис. 2.4. Хотя в примере с прямоугольником дублируются всего две вершины, по мере роста детализированности и сложности модели число таких вершин быстро растет. Например, у изображенного на рис. 2.5 куба восемь уникальных вершин, но в списке треугольников, образующих куб каждая из этих вершин встречается по несколько раз.

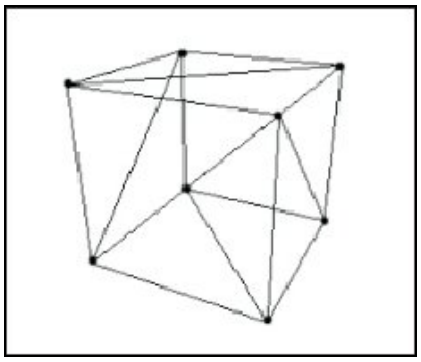

Рис. 2.5. *Составленный из треугольников куб*

Для решения этой проблемы мы добавим концепцию *индексов* (*indices*). Она действует следующим образом: мы создаем список вершин и список индексов. В списке вершин перечисляются все уникальные вершины, а список индексов содержит последовательность номеров (индексов) вершин из списка вершин, показывающую как объединяются вершины для формирования треугольников. Для примера с прямоугольником список вершин мог бы выглядеть так:

```
Vertex vertexList[4] = {v0, v1, v2, v3};
```

Тогда список индексов, описывающий как из имеющихся вершин формируются два треугольника, будет выглядеть так:

```
WORD indexList[6] = {0, 1, 2,  // треугольник 0
                     0, 2, 3}; // треугольник 1
```

Если облечь код в слова, определение массива **`indexList`** говорит, что треугольник 0 образован нулевым (**`vertexList[0]`**), первым (**`vertexList[1]`**) и вторым (**`vertexList[2]`**) элементами списка вершин, а треугольник 1 образован нулевым (**`vertexList[0]`**), вторым (**`vertexList[2]`**) и третьим (**`vertexList[3]`**) элементами списка вершин.

2.2 Виртуальная камера

Камера определяет какую часть мира может видеть зритель и, следовательно, для какой части мира нам надо создавать ее двухмерное изображение. Камера позиционируется и ориентируется в пространстве и определяет видимую облать пространства. Схема используемой нами модели камеры показана на рис. 2.6.

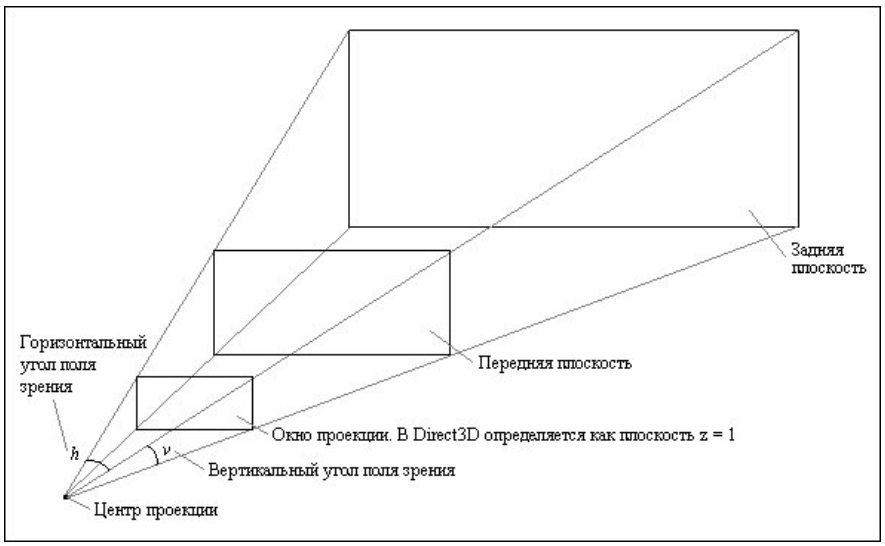

Рис. 2.6. *Усеченная пирамида, определяющая область пространства, которую «видит» камера*

Область видимого пространства преставляет собой *усеченную пирамиду* (*frustum*) и определяется углами поля зрения, передней и задней плоскостями. Причины использования усеченной пирамиды станут ясны, если принять во внимание, что экран на котором отображается сцена — прямоугольный. Объекты, которые не находятся внутри заданного пространства невидимы и должны быть исключены из процесса дальнейшей обработки. Процесс исключения таких данных называется *отсечением* (*clipping*).

Окно проекции (*projection window*) — это двухмерная область на которую проецируются находящиеся внутри области видимого пространства трехмерные объекты для создания двухмерного изображения, представляющего трехмерную сцену. Важно помнить, что мы определяем окно проекции таким образом, что координаты его правого верхнего угла будут (1, 1), а координаты левого нижнего угла — (–1, –1).

Для упрощения рисования в программах из этой книги *плоскость проекции* (плоскость в которой расположено окно проекции) и передняя плоскость совпадают. Также обратите внимание, что Direct3D определяет плоскость проекции как плоскость $z = 1$.

2.3 Конвейер визуализации

Как только мы описали геометрию трехмерной сцены и установили виртуальную камеру, перед нами встает задача формирования двухмерного представления этой сцены на мониторе. Последовательность действий, которые должны быть выполнены для решения этой задачи называется *конвейером визуализации* (*rendering pipeline*). На рис. 2.7 представлена упрощенная схема этого конвейера и в последующих разделах мы подробнее обсудим каждый его этап.

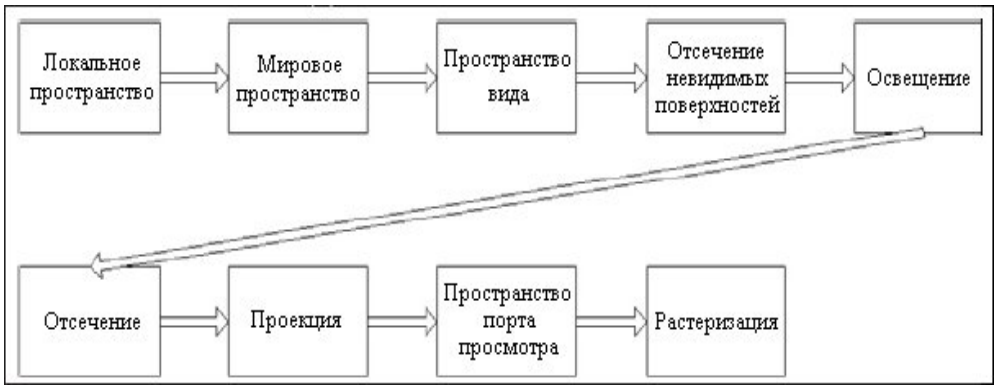

Рис. 2.7. *Упрощенная схема конвейера визуализации*

Несколько этапов конвейера выполняют преобразование из одной системы координат в другую. Эти преобразования выполняются с помощью матриц. Direct3D выполняет вычисления преобразований за нас. Это полезно, потому что преобразования могут выполняться аппаратурой, если ваша видеокарта

поддерживает аппаратную обработку преобразований. Если мы используем для преобразований Direct3D, нам надо только предоставить матрицу преобразования, которая описывает преобразования, необходимые для перехода от одной системы координат к другой. Мы задаем матрицу с помощью метода **`IDirect3DDevice->SetTransform`**. Он получает параметр, описывающий тип преобразования и указатель на матрицу преобразования. Например, на рис. 2.7, для преобразования, необходимого для перехода от локального пространства к мировому, мы должны написать:

```
Device->SetTransform(D3DTS_WORLD, &worldMatrix);
```

В последующих разделах, где исследуется каждый этап конвейера визуализации, мы узнаем больше об этом методе.

2.3.1 Локальное пространство

Локальное пространство (*local space*) или *пространство моделирования* (*modeling space*) — это та система координат, в которой мы описываем объект в виде списка треугольных граней. Локальное пространство полезно потому что оно упрощает процесс моделирования. Создавать модель в ее собственной, локальной системе координат проще чем встраивать ее непосредственно в сцену. Локальное пространство позволяет нам создавать модели не заботясь об их расположении, размере или ориентации относительно других объектов сцены.

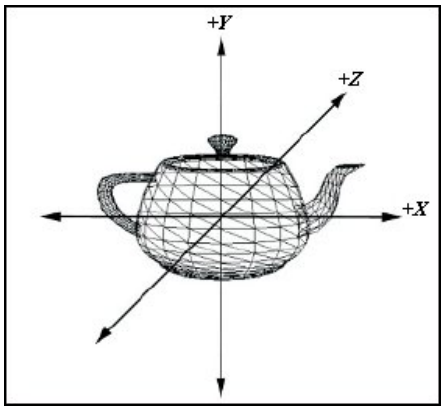

Рис. 2.8. *Чайник, описанный в его собственной, локальной системе координат*

2.3.2 Мировое пространство

После того, как мы создали различные модели, каждая из которых описана в своей собственной локальной системе координат, нам надо собрать их воедино в сцену, описанную в единой, глобальной (мировой) системе координат (world space). Объекты преобразуются из локального пространства в мировое с помощью процесса, называемого *мировым преобразованием* (*world transform*), который обычно состоит из операций перемещения, вращения и масштабирования в результате которых модель приобретает то местоположние, ориентацию и размеры, которые должны быть у нее в сцене. Мировое преобразование задает

взаимосвязь между всеми объектами мира в части их местоположения, размера и ориентации.

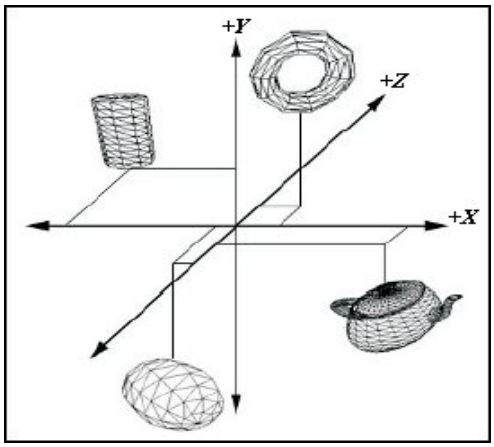

Рис. 2.9. *Несколько трехмерных объектов, описанных в единой мировой системе координат*

Мировое преобразование представляется с помощью матрицы и устанавливается в Direct3D с помощью метода **`IDirect3DDevice9::SetTransform`**, где в качестве вида преобразования указано **`D3DTS_WORLD`**. Предположим, мы хотим поместить куб в точку (–3, 2, 6) мирового пространства, а сферу — в точку (5, 0, –2). Для этого следует написать:

```
// Создаем матрицу мирового преобразования для куба,
// которая содержит только перемещение
D3DXMATRIX cubeWorldMatrix;
D3DXMatrixTranslation(&cubeWorldMatrix, -3.0f, 2.0f, 6.0f);

// Создаем матрицу мирового преобразования для сферы,
// которая содержит только перемещение
D3DXMATRIX sphereWorldMatrix;
D3DXMatrixTranslation(&sphereWorldMatrix, 5.0f, 0.0f, -2.0f);

// Устанавливаем преобразование для куба
Device->SetTransform(D3DTS_WORLD, &cubeWorldMatrix);
drawCube(); // рисуем куб

// Теперь, поскольку сфера использует другую матрицу мирового
// преобразования, мы должны изменить мировое преобразование для
// сферы. Если не сделать этого, сфера будет рисоваться с
// использованием предыдущей матрицы мирового преобразования,
// которая предназначалась для куба.
Device->SetTransform(D3DTS_WORLD, &sphereWorldMatrix);
drawSphere(); // рисуем сферу
```

Это очень упрощенный пример, поскольку обычно объекты приходится не только перемещать, но и вращать и масштабировать, но он показывает как работает мировое преобразование.

2.3.3 Пространство вида

В мировом пространстве геометрия объектов и камера описаны относительно одной мировой системы координат, как показано на рис. 2.10. Однако, проекция и другие операции станут более трудными и менее эффективными, если камера не занимает определенное местоположение и не ориентирована требуемым образом. Чтобы упростить вычисления мы перемещаем камеру в начало координат и поворачиваем ее таким образом, чтобы она была направлена вдоль положительного направления оси Z. Вместе с камерой перемещаются и поворачиваются и все образующие сцену объекты, так что вид сцены остается неизменным. Данное преобразование называется *преобразованием пространства вида* (*view space transformation*), и после него говорят, что объекты расположены в *пространстве вида* (*view space*).

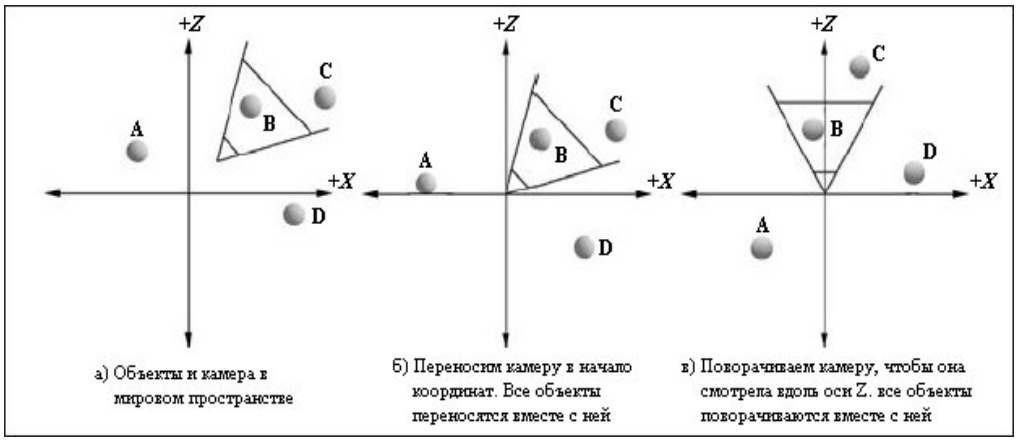

Рис. 2.10. *Преобразование из мирового пространства в пространство вида. В результате этого преобразования камера перемещается в начало координат и поворачивается так, чтобы быть направленной вдоль положительного направления оси Z. Обратите внимание, что все объекты сцены также подвергаются этому преобразованию, так что формируемый камерой вид сцены не изменяется*

Чтобы вычислить матрицу преобразования вида можно воспользоваться следующей функцией библиотеки D3DX:

```
D3DXMATRIX *D3DXMatrixLookAtLH(
    D3DXMATRIX*        pOut, // указатель на возвращаемую
                             // матрицу преобразования
    CONST D3DXVECTOR3* pEye, // местоположение камеры в сцене
    CONST D3DXVECTOR3* pAt,  // точка, на которую направлена камера
    CONST D3DXVECTOR3* pUp   // вектор, задающий направление
                             // вверх – (0, 1, 0)
);
```

Параметр **`pEye`** задает точку пространства, в которой располагается камера. Параметр **`pAt`** задает ту точку сцены, на которую направлена камера. Параметр

pUp — это вектор, который задает направление «вверх» для нашей сцены. Почти всегда это вектор, совпадающий с осью Y — (0, 1, 0).

Предположим, мы разместили камеру в точке (5, 3, –10) и направили ее на начало системы координат нашей сцены (0, 0, 0). Чтобы создать матрицу преобразования пространства вида, надо написать:

```
D3DXVECTOR3 position(5.0f, 3.0f, -10.0f);
D3DXVECTOR3 targetPoint(0.0f, 0.0f, 0.0f);
D3DXVECTOR3 worldUp(0.0f, 1.0f, 0.0f);

D3DXMATRIX V;
D3DXMatrixLookAtLH(&V, &position, &targetPoint, &worldUp);
```

Преобразование пространства вида устанавливается с помощью метода **IDirect3DDevice9::SetTransform**, у которого в качестве типа преобразования указано **D3DTS_VIEW**:

```
Device->SetTransform(D3DTS_VIEW, &V);
```

2.3.4 Удаление невидимых поверхностей

У полигона есть две стороны; одну из них мы будем называть *лицевой* (*front*), а другую — *обратной* (*back*). Обычно обратные стороны полигонов никогда не видны. Это происходит из-за того, что большинство объектов сцены являются сплошными объемными телами, такими как ящики, цилиндры, цистерны, персонажи и т.д. и камера никогда не попадает внутрь занимаемого объектом пространства. Поэтому камера никогда не может увидеть обратные стороны полигонов. Это очень важно знать, потому что если мы дадим возможность видеть обратные стороны полигонов, удаление невидимых поверхностей не будет работать.

На рис. 2.11 показан объект в пространстве вида, лицевая сторона каждой грани которого помечена стрелкой. Полигон, лицевая сторона которого обращена к камере, называется *фронтальным полигоном* (*front facing polygon*), а полигон, лицевая сторона которого обращена от камеры, называется *обратным полигоном* (*back facing polygon*).

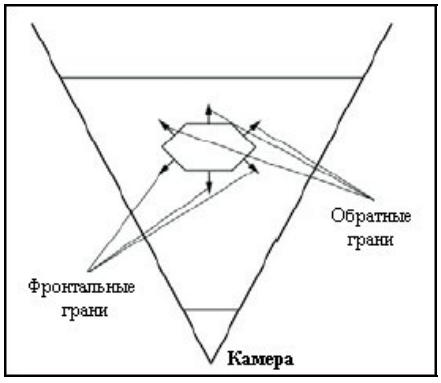

Рис. 2.11. *Объект с фронтальными и обратными полигонами*

Исследовав рис. 2.11 мы увидим, что фронтальные полигоны скрывают находящиеся за ними обратные полигоны. Direct3D может извлечь из этого пользу отбросив (исключив из дальнейшей обработки) обратные полигоны; этот процесс называется *удалением невидимых граней* (*backface culling*). На рис. 2.12 показан тот же самый объект, но уже после удаления невидимых граней. Камера будет все равно показывать ту же самую сцену, поскольку обратные грани скрыты и в любом случае их нельзя увидеть.

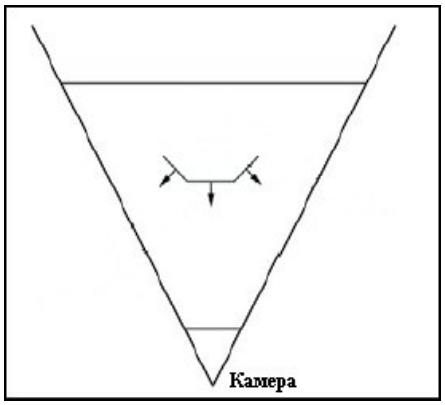

Рис. 2.12. *Сцена после отбрасывания обратных граней*

Конечно, чтобы выполнить эту работу, Direct3D необходимо знать, какой полигон является фронтальным, а какой — обратным. По умолчанию Direct3D считает фронтальными гранями те треугольники, у которых порядок обхода вершин (в пространстве вида) задан по часовой стрелке. Треугольники, у которых порядок обхода вершин задан против часовой стрелки (опять же в пространстве вида) считаются обратными гранями.

ПРИМЕЧАНИЕ Обратите внимание, что мы говорим «в пространстве вида». Это вызвано тем, что при повороте треугольника на 180 градусов порядок обхода вершин у него меняется на противоположный. Следовательно, треугольник у которого порядок обхода вершин в локальном пространстве был по часовой стрелке, может сменить порядок обхода вершин в пространстве вида, поскольку в процессе преобразований выполнялось вращение объектов.

Если по каким-то причинам вас не устраивает принятый по умолчанию вариант отбрасывания обратных граней, можно изменить его путем изменения режима визуализации **D3DRS_CULLMODE**.

```
Device->SetRenderState(D3DRS_CULLMODE, Value);
```

где **Value** может принимать одно из следующих значений:

- **D3DCULL_NONE** — Удаление обратных граней выключено.
- **D3DCULL_CW** — Отбрасываются треугольники с порядком обхода вершин по часовой стрелке.
- **D3DCULL_CCW** — Отбрасываются треугольники с порядком обхода вершин против часовой стрелки. Это значение по умолчанию.

2.3.5 Освещение

Источники света описываются в мировом пространстве, но потом преобразуются в пространство вида при соответствующем преобразовании сцены. В пространстве вида источники света применяются для освещения объектов сцены, что позволяет получить более реалистичный вид. Работа с освещением в фиксированном конвейере визуализации подробно рассматривается в главе 5. Позднее, в части IV, мы реализуем собственную схему освещения с использованием программируемого конвейера.

2.3.6 Отсечение

Теперь нам необходимо отбросить геометрию, которая находится вне видимого пространства; этот процесс называется *отсечением* (*clipping*). Есть три варианта размещения треугольной грани относительно усеченной пирамиды видимого пространства:

- Полностью внутри — Если треугольник полностью находится внутри области видимого пространства, он переходит на следующий этап.
- Полностью снаружи — если треугольник находится полностью вне пирамиды видимого пространства, он исключается из процесса дальнейшей обработки.
- Частично внутри (частично снаружи) — если внутри пирамиды видимого пространства находится только часть треугольника, то он разбивается на две части. Часть, которая находится внутри пирамиды видимого пространства остается, а часть, которая находится снаружи — отбрасывается.

Все три рассмотренных варианта изображены на рис. 2.13.

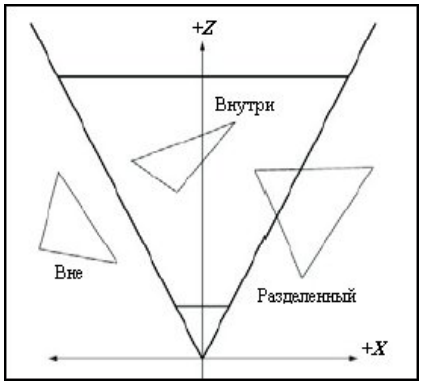

Рис. 2.13. *Отсечение геометрии, находящейся вне видимого пространства*

2.3.7 Проекция

Для пространства вида остается задача получения двухмерного представления трехмерной сцены. Процесс перехода от *n*-мерного пространства к (*n*–1)-мерному называется *проекцией* (*projection*). Существует множество способов выполнить

проекцию, но нас интересует один частный случай, называемый *перспективной проекцией* (*perspective projection*). Перспективная проекция выполняется таким образом, что объекты, расположенные дальше от камеры выглядят меньше, чем объекты того же размера, находящиеся ближе к камере. Этот тип проекции позволяет представить трехмерную сцену в виде двухмерного изображения. На рис. 2.14 показана точка в трехмерном пространстве, проецируемая в на плоскость проекции с использованием перспективной проекции.

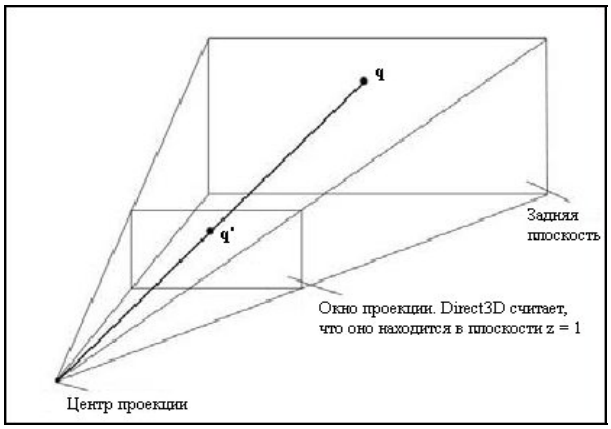

Рис. 2.14. *Проекция точки в трехмерном пространстве в окно проекции*

Преобразование проекции описывает наше видимое пространство (усеченную пирамиду) и отвечает за проецирование геометрии из него в окно проекции. Матрица проекции сложная и мы не будем обсуждать формулы для ее получения. Вместо этого воспользуемся следующей функцией библиотеки D3DX, которая создает матрицу проекции на основании описания усеченной пирамиды видимого пространства.

```
D3DXMATRIX *D3DXMatrixPerspectiveFovLH(
    D3DXMATRIX* pOut, // возвращает матрицу проекции
    FLOAT fovY,       // вертикальный угол поля зрения в радианах
    FLOAT Aspect,     // форматное соотношение = ширина / высота
    FLOAT zn,         // расстояние до передней полскости
    FLOAT zf          // расстояние до задней плоскости
);
```

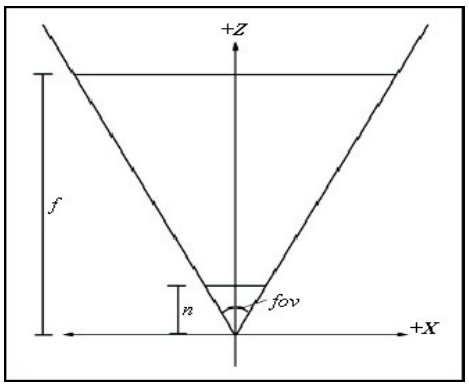

Рис. 2.15. *Компоненты пирамиды видимого пространства*

Параметр форматного соотношения заслуживает дополнительных пояснений. Геометрия в окне проекции в конечном итоге преобразуется в экранное пространство (см. раздел 2.3.8). Перенос изображения из квадратной области (окна проекции) на экран, который, как известно, прямоугольный, приводит к возникновению искажений. Форматное соотношение, являющееся просто соотношением между сторонами экрана, используется для корректировки искажений возникающих при отображении квадрата на прямоугольник.

форматноеСоотношение = ширинаЭкрана / высотаЭкрана

Матрица проекции устанавливается с помощью метода **`IDirect3DDevice9::SetTransform`**, в котором указан тип преобразования **`D3DTS_PROJECTION`**. В приведенном ниже примере создается матрица проекции на основании усеченной пирамиды видимого пространства с углом поля зрения в 90 градусов, передней плоскостью, расположенной на расстоянии 1 единицы и задней плоскостью, расположенной на расстоянии в 1000 единиц.

```
D3DXMATRIX proj;
D3DXMatrixPerspectiveFovLH(
    &proj, PI * 0.5f, (float)width / (float)height, 1.0, 1000.0f);
Device->SetTransform(D3DTS_PROJECTION, &proj);
```

ПРИМЕЧАНИЕ Заинтересованные читатели могут подробнее узнать о проекции во втором издании книги Алана Ватта «3D Computer Graphics».

2.3.8 Преобразование порта просмотра

Преобразование порта просмотра отвечает за преобразование координат из окна проекции в прямоугольную область экрана, которая называется *портом просмотра* (*viewport*). Для игр портом просмотра обычно является весь экран. Однако, если приложение работает в оконном режиме, портом просмотра является часть экрана или клиентская область. Прямоугольник порта просмотра описывается относительно содержащего его окна и задается в оконных координатах (рис. 2.16).

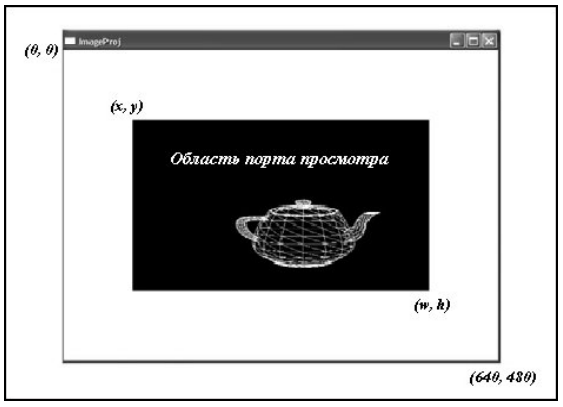

Рис. 2.16. *Прямоугольник порта просмотра*

В Direct3D порт просмотра представляется структурой **D3DVIEWPORT9**. Ее объявление выглядит так:

```
typedef struct _D3DVIEWPORT9 {
    DWORD X;
    DWORD Y;
    DWORD Width;
    DWORD Height;
    DWORD MinZ;
    DWORD MaxZ;
} D3DVIEWPORT9;
```

Первые четыре члена данных описывают прямоугольник порта просмотра относительно содержащего его окна. Переменная **MinZ** задает минимальное значение буфера глубины, а переменная **MaxZ** — максимальное значение буфера глубины. Direct3D использует значения буфера глубины в диапазоне от нуля до единицы, поэтому переменным **MinZ** и **MaxZ** следует присваивать эти значения, если только вы не хотите реализовать какие-нибудь спецэффекты.

После инициализации структуры **D3DVIEWPORT9**, мы устанавливаем порт просмотра Direct3D следующим образом:

```
D3DVIEWPORT9 vp{ 0, 0, 640, 480, 0, 1 };
Device->SetViewport(&vp);
```

Direct3D выполняет преобразование порта просмотра за нас автоматически, но для справки мы приведем матрицу, описывающую это преобразование. Переменные в ней соответствуют одноименным членам данных структуры **D3DVIEWPORT9**.

$$\begin{bmatrix} \frac{Width}{2} & 0 & 0 & 0 \\ 0 & -\frac{Height}{2} & 0 & 0 \\ 0 & 0 & MaxZ - MinZ & 0 \\ X + \frac{Width}{2} & Y + \frac{Height}{2} & MinZ & 1 \end{bmatrix}$$

2.3.9 Растеризация

После преобразования вершин в экранные координаты, у нас образуется список двухмерных треугольников. Этап растеризации отвечает за вычисление цветов отдельных пикселей, образующих треугольник (рис. 2.17).

Процесс растеризации требует выполнения огромного объема вычислений в выполнение которых всегда вовлекается процессор видеокарты. Конечным результатом этапа растеризации является двухмерное изображение, выводимое на экран монитора.

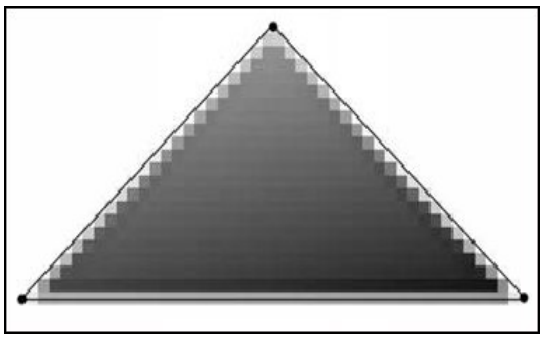

Рис. 2.17. *Растеризация треугольника на экране*

2.4 Итоги

- Трехмерные объекты представляются с помощью сетки из треугольных ячеек — списка треугольников, описывающих форму и контуры объекта.
- Моделью виртуальной камеры является усеченная пирамида. Пространство внутри приамиды — это то, что «видит» камера.
- Трехмерные объекты описываются каждый в своем локальном пространстве, а затем все переносятся в общее, мировое пространство. Чтобы упростить проекцию, отбрасывание невидимых граней и другие операции, объекты переносятся в пространство вида, в котором камера расположена в начале координат и направлена вдоль положительного направления оси Z. После преобразования в пространство вида выполняется проекция объектов в окно проекции. Преобразование порта просмотра переносит геометрию из окна проекции в область порта просмотра. И, в самом конце, на этапе растеризации вычисляется цвет каждого пикселя итогового двухмерного изображения.

Глава 3
Рисование в Direct3D

В предыдущей главе мы изучили основные концепции создания и визуализации трехмерных сцен. В этой главе мы применим полученные знания на практике и узнаем как в Direct3D можно рисовать геометрические объекты. Рассматриваемые в этой главе интерфейсы и методы Direct3D очень важны, так как будут использоваться на протяжении всей оставшейся части кинги.

Цели

- Узнать, как в Direct3D хранятся данные вершин и индексов.
- Изучить, как можно менять способ отображения фигур с помощью режимов визуализации.
- Научиться визуализировать сцены.
- Узнать, как с помощью функций **`D3DXCreate*`** можно создавать сложные трехмерные объекты.

3.1 Буферы вершин и индексов

Буферы вершин и индексов обладают сходными интерфейсами и предоставляют одинаковые методы, так что мы будем их рассматривать вместе. Буфер вершин представляет собой непрерывный участок памяти в котором хранятся данные вершин. Аналогичным образом, буфер индексов — это непрерывный участок памяти в котором хранятся данные индексов. Мы используем буферы вершин и индексов для хранения данных из соответствующих массивов по той причине, что эти буферы могут располагаться в памяти видеокарты. Визуализация данных, находящихся в памяти видеокарты, выполняется гораздо быстрее, чем визуализация данных, расположенных в системной памяти.

В коде буфер вершин представляется интерфейсом **`IDirect3DVertexBuffer9`**, а буфер индексов представляется интерфейсом **`IDirect3DIndexBuffer9`**.

3.1.1 Создание буферов вершин и индексов

Для создания буферов вершин и индексов используются следующие два метода:

```
HRESULT IDirect3DDevice9::CreateVertexBuffer(
     UINT Length,
     DWORD Usage,
     DWORD FVF,
     D3DPOOL Pool
     IDirect3DVertexBuffer9** ppVertexBuffer,
     HANDLE* pSharedHandle
);

HRESULT IDirect3DDevice9::CreateIndexBuffer(
     UINT Length,
     DWORD Usage,
     D3DFORMAT Format,
     D3DPOOL Pool,
     IDirect3DIndexBuffer9** ppIndexBuffer,
     HANDLE* pSharedHandle
);
```

У обоих методов большая часть параметров идентична, так что рассмотрим параметры обоих методов вместе:

- **`Length`** — Количество байт, выделяемых под буфер. Если нам требуется буфер достаточного размера для хранения восьми вершин, в этом параметре следует указать **`8 * sizeof(Vertex)`**, где **`Vertex`** — это ваша структура данных вершины.
- **`Usage`** — Задает ряд дополнительных параметров, определяющих особенности использования буфера. Можно указать 0, если

дополнительные параметры отсутствуют, или комбинацию из одного или нескольких следующих флагов:

- **D3DUSAGE_DYNAMIC** — Флаг указывает, что данный буфер будет динамическим. Различия между статическими и динамическими буферами мы обсудим чуть позже.
- **D3DUSAGE_POINTS** — Флаг указывает, что буфер будет использовваться для хранения примитивов точек. Примитивы точек рассматриваются в главе 14. Этот флаг может применяться только для буферов вершин.
- **D3DUSAGE_SOFTWAREPROCESSING** — Обработка вершин будет выполняться программно.
- **D3DUSAGE_WRITEONLY** — Указывает, что приложение может только записывать данные в буфер. Это позволяет драйверу разместить буфер в области памяти, обеспечивающей максимальную скорость записи. Обратите внимание, что попытка прочитать данные из буфера, созданного с указанием этого флага, приведет к ошибке.

- **FVF** — Настраиваемый формат вершин, которые будут храниться в создаваемом буфере вершин.
- **Pool** — Пул памяти, в котором будет размещен буфер.
- **ppVertexBuffer** — Адрес ля возврата указателя на созданный буфер вершин.
- **pSharedHandle** — Не используется, должен быть равен 0.
- **Format** — Задает размер индексов; используйте **D3DFMT_INDEX16** для 16-разрядных индексов или **D3DFMT_INDEX32** для 32-разрядных индексов. Обратите внимание, что не все устройства поддерживают 32-разрядные индексы, не забывайте проверять возможности устройства.
- **ppIndexBuffer** — Адрес для возврата указателя на созданный буфер индексов.

ПРИМЕЧАНИЕ Буфер, созданный *без* указания флага **D3DUSAGE_DYNAMIC** называется *статическим буфером*. Обычно статический буфер располагается в видеопамяти, где его содержимое может обрабатываться более эффективно. Однако, если вы сделаете буфер статическим, придется расплачиваться исключительно низкой скоростью записи в буфер и чтения из него. Поэтому мы будем использовать статические буферы для хранения статических данных (данных, которые не надо часто изменять). Хорошими кандидатами на размещение в статическом буфере являются ландшафты и здания, поскольку их геометрия обычно не изменяется во время работы приложения. Данные геометрии должны заноситься в статические буферы во время инициализации приложения, а не во время его работы.

ПРИМЕЧАНИЕ Буфер, созданный *с* указанием флага **D3DUSAGE_DYNAMIC** называется *динамическим буфером*. Динамические буферы обычно располагаются в памяти AGP, где их содержимое может достаточно быстро обновляться. Видеокарта работает с динамическими буферами медленнее, чем со статическими, потому что данные перед визуализацией должны быть переданы из буфера в видеопамять. Главное преимущество динамических буферов — возможность быстрого обновления (быстрая запись данных центральным процессором). Следовательно, если вам надо часто изменять содержимое буфера, сделайте его динамическим. Хорошим кандидатом на размещение в динамическом буфере являются системы частиц, поскольку они анимируются и их геометрия обычно изменяется в каждом кадре.

ПРИМЕЧАНИЕ Чтение данных из видеопамяти или из памяти AGP выполняется очень медленно. Следовательно, если вам во время работы приложения надо читать данные геометрии, лучше всего создать копию данных в системной памяти и обращаться для чтения к ней.

В приведенном ниже фрагменте кода создается статический буфер вершин в котором можно хранить до восьми вершин типа **Vertex**.

```
IDirect3DVertexBuffer9* vb;
_device->CreateVertexBuffer(
               8 * sizeof(Vertex),
               0,
               D3DFVF_XYZ,
               D3DPOOL_MANAGED,
               &vb,
               0);
```

Следующий фрагмент кода показывает как создать динамический буфер индексов, в котором можно хранить до 36 16-разрядных индексов.

```
IDirect3DIndexBuffer9* ib;
_device->CreateIndexBuffer(
               36 * sizeof(WORD),
               D3DUSAGE_DYNAMIC | D3DUSAGE_WRITEONLY,
               D3DFMT_INDEX16,
               D3DPOOL_MANAGED,
               &ib,
               0);
```

3.1.2 Доступ к памяти буфера

Для доступа к памяти буфера вершин или индексов нам необходимо получить указатель на область памяти с содержимым буфера. Мы получаем указатель на содержимое с помощью метода **Lock**. Не забудьте после завершения работы с

буфером разблокировать его. Получив указатель на область памяти, можно считывать и записывать информацию.

ПРИМЕЧАНИЕ Если при создании буфера вершин или индексов был указан флаг **D3DUSAGE_WRITEONLY**, вы не сможете читать информацию из этого буфера. Попытка выполнить чтение приведет к возникновению ошибки.

```
HRESULT IDirect3DVertexBuffer9::Lock(
        UINT OffsetToLock,
        UINT SizeToLock,
        BYTE** ppbData,
        DWORD Flags
);

HRESULT IDirect3DIndexBuffer9::Lock(
        UINT OffsetToLock,
        UINT SizeToLock,
        BYTE** ppbData,
        DWORD Flags
);
```

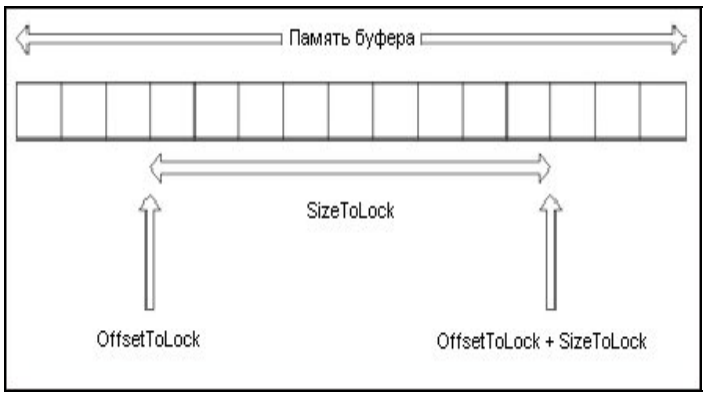

Рис. 3.1. *Параметры* ***OffsetToLock*** *и* ***SizeToLock*** *определяют блокируемую область памяти. Если обоим параметрам присвоить нулевые значения, будет заблокирован весь буфер*

Параметры у обоих методов одинаковы.

- **OffsetToLock** — смещение в байтах от начала буфера до начала блокируемой области (рис. 3.1).
- **SizeToLock** — количество блокируемых байтов.
- **ppbData** — Адрес для возврата указателя на начало заблокированной области памяти.
- **Flags** — Флаги, задающие режим блокировки. Можно указать ноль, либо комбинацию из одного или нескольких перечисленных ниже флагов:
    - **D3DLOCK_DISCARD** — Этот флаг используется только для динамических буферов. Он приказывает аппаратуре оставить старый буфер и возвратить указатель на новый буфер, который должен быть создан. Данная возможность полезна потому что позволяет аппаратуре продолжать визуализацию с

использованием старого буфера в то время как пользователь работает с новым буфером. Это предотвращает простои оборудования.

- **D3DLOCK_NOOVERWRITE** — Данный флаг используется только для динамических буферов. Он сообщает, что вы будете только добавлять данные в буфер. Это значит, что вы не будете перезаписывать никакие данные, которые уже используются для визуализации. Следовательно видеокарта может продолжать выполнять визуализацию, в то время когда вы добавляете новые данные в буфер.
- **D3DLOCK_READONLY** — Этот флаг указывает, что вы хотите заблокировать буфер только для чтения и ничего не будете записывать в него. Благодаря этому система может выполнить внутреннюю оптимизацию.

Флаги **D3DLOCK_DISCARD** и **D3DLOCK_NOOVERWRITE** опираются на тот факт, что в момент вызова функции блокировки часть памяти буфера может использоваться для визуализации. Если обстоятельства позволяют указывать эти флаги, то благодаря их использованию можно избежать простоев визуализации, которые могли бы произойти в ином случае.

Приведенный ниже пример демонстрирует обычный вариант использования метода **Lock**. Обратите внимание, что закончив работу мы сразу вызываем метод **Unlock**.

```
Vertex* vertices;
_vb->Lock(0, 0, (void**)&vertices, 0);    // заблокировать весь буфер

vertices[0] = Vertex(-1.0f, 0.0f, 2.0f);  // записать данные вершин
vertices[1] = Vertex( 0.0f, 1.0f, 2.0f);  // в буфер
vertices[2] = Vertex( 1.0f, 0.0f, 2.0f);

_vb->Unlock();                            // разблокировать буфер, когда
                                          // мы закончили работать с ним
```

3.1.3 Получение информации о буфере

Иногда нам может потребоваться получить информацию о буфере вершин или индексов. Приведенный ниже фрагмент кода показывает методы, используемые для получения этой информации:

```
D3DVERTEXBUFFER_DESC vbDescription;
_vertexBuffer->GetDesc(&vbDescription); // получаем информацию
                                         // о буфере вершин

D3DINDEXBUFFER_DESC ibDescription;
_indexBuffer->GetDesc(&ibDescription); // получаем информацию
                                       // о буфере индексов
```

Объявление структур **D3DVERTEXBUFFER_DESC** и **D3DINDEXBUFFER_DESC** выглядит следующим образом:

```
typedef struct _D3DVERTEXBUFFER_DESC {
     D3DFORMAT Format;
     D3DRESOURCETYPE Type;
     DWORD Usage;
     D3DPOOL Pool;
     UINT Size;
     DWORD FVF;
} D3DVERTEXBUFFER_DESC;

typedef struct _D3DINDEXBUFFER_DESC {
     D3DFORMAT Format;
     D3DRESOURCETYPE Type;
     DWORD Usage;
     D3DPOOL Pool;
     UINT Size;
} D3DINDEXBUFFER_DESC;
```

3.2 Режимы визуализации

В Direct3D включен набор *режимов визуализации* (*rendering state*), которые влияют на способ рисования объектов. У каждого режима визуализации есть значение по умолчанию, так что вам надо менять их только в том случае, если требуемый приложению режим визуализации отличается от предлагаемого по умолчанию. Установленный режим визуализации влияет на процесс рисования до тех пор, пока вы снова не измените его. Для установки режимов визуализации используется следующий метод:

```
HRESULT IDirect3DDevice9::SetRenderState(
     D3DRENDERSTATETYPE State, // изменяемый режим
     DWORD Value               // новое значение режима
);
```

Например, в примерах из этой главы мы выполняем визуализацию объектов в каркасном режиме. Поэтому нам необходимо установить следующий режим визуализации:

```
_device->SetRenderState(D3DRS_FILLMODE, D3DFILL_WIREFRAME);
```

ПРИМЕЧАНИЕ Чтобы увидеть все доступные режимы визуализации, посмотрите описание типа **D3DRENDERSTATETYPE** в DirectX SDK.

3.3 Подготовка к рисованию

Создав буфер вершин и, возможно, буфер индексов мы уже почти готовы к визуализации их содержимого, но сперва надо выполнить несколько подготовительных действий.

1. Установка источника потоковых данных. Эта операция подключает буфер вершин к потоку, основная задача которого — снабжение конвейера визуализации данными о геометрии.

   Для установки источника потоковых данных используется следующий метод:

   ```
   HRESULT IDirect3DDevice9::SetStreamSource(
        UINT StreamNumber,
        IDirect3DVertexBuffer9* pStreamData,
        UINT OffsetInBytes,
        UINT Stride
   );
   ```

   - **StreamNumber** — Идентифицирует поток, к которому мы будем подключать буфер вершин. В этой книге мы не используем несколько потоков, так что значение данного параметра всегда будет равно нулю.
   - **pStreamData** — Указатель на буфер вершин,который мы хотим подключить к потоку.
   - **OffsetInBytes** — Измеренное в байтах смещение от начала потока, задающее начало данных вершин, которые будут переданы в конвейер визуализации. Если вы собираетесь указывать отличное от нуля значение, убедитесь, что устройство поддерживает данную возможность, проверив установлен ли флаг **D3DDEVCAPS2_STREAMOFFSET** в структуре **D3DCAPS9**.
   - **Stride** — Размер в байтах каждого элемента того буфера вершин, который мы подключаем к потоку.

   Предположим, что **vb** — это буфер вершин, заполненный данными вершин типа **Vertex**. В этом случае обращение к методу будет выглядеть так:

   ```
   _device->SetStreamSource(0, vb, 0, sizeof(Vertex));
   ```

2. Задание формата вершин. Здесь мы указываем формат вершин, который будет использоваться в дальнейших вызовах функций рисования.

   ```
   _device->SetFVF(D3DFVF_XYZ | D3DFVF_DIFFUSE | D3DFVF_TEX1);
   ```

3. Задание буфера индексов. Если мы используем буфер индексов, то должны указать тот буфер индексов, который будет использоваться в последующих операциях рисования. Одновременно может использоваться только один буфер индексов; следовательно, если вам потребуется

нарисовать объект с использованием другого буфера индексов, надо будет переключиться на другой буфер. Задание буфера индексов выполняет следующий фрагмент кода:

```
_device->SetIndices(_ib); // передаем копию указателя
                          // на буфер индексов
```

3.4 Рисование с буферами вершин и индексов

После того, как мы создали наши буферы вершин и индексов и выполнили все подготовительные действия, можно рисовать наши фигуры, передавая данные об их геометрии в конвейер визуализации с помощью методов **DrawPrimitive** или **DrawIndexedPrimitive**. Эти методы получают информацию о вершинах из настроенного потока вершин и индексы из установленного в данный момент буфера индексов.

3.4.1 IDirect3DDevice9::DrawPrimitive

Данный метод используется для рисования примитивов не использующих индексы.

```
HRESULT IDirect3DDevice9::DrawPrimitive(
     D3DPRIMITIVETYPE PrimitiveType,
     UINT StartVertex,
     UINT PrimitiveCount
);
```

- **PrimitiveType** — Тип рисуемого примитива. Помимо треугольников вы можете рисовать линии и точки. Поскольку мы используем треугольники, в данном параметре следует указать **D3DPT_TRIANGLELIST**.
- **StartVertex** — Индекс элемента потока вершин с которого следует начать чтение данных вершин. Благодаря этому параметру мы можем рисовать только часть буфера вершин.
- **PrimitiveCount** — Количество рисуемых примитивов.

А вот пример использования метода:

```
// рисуем четыре треугольника
_device->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 4);
```

3.4.2 IDirect3DDevice9::DrawIndexedPrimitive

Этот метод используется для рисования примитивов, использующих индексы.

```
HRESULT IDirect3DDevice9::DrawIndexedPrimitive(
     D3DPRIMITIVETYPE Type,
     INT BaseVertexIndex,
     UINT MinIndex,
     UINT NumVertices,
     UINT StartIndex,
     UINT PrimitiveCount
);
```

- **Type** — Тип рисуемого примитива. Помимо треугольников вы можете рисовать линии и точки. Поскольку мы используем треугольники, в данном параметре следует указать **D3DPT_TRIANGLELIST**.
- **BaseVertexIndex** — Базисная величина, которая будет прибавлена к используемым в данном вызове индексам. Подробнее о ней говорится в находящемся после списка примечании.
- **MinIndex** — Минимальное значение индекса, которое будет использовано.
- **NumVertices** — Количество вершин, которые будут обработаны данным вызовом.
- **StartIndex** — Номер элемента буфера индексов, который будет отмечен как стартовая точка с которой начнется чтение индексов.
- **PrimitiveCount** — Количество рисуемых примитивов.

Пример использования метода:

```
_device->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0, 8, 0, 12);
```

ПРИМЕЧАНИЕ Параметр **BaseVertexIndex** заслуживает более подробного исследования. В исследовании вам поможет рис. 3.2. Локальные буферы индексов ссылаются на вершины в соответствующих локальных буферах вершин. Теперь представьте, что мы объединили вершины сферы, куба и цилиндра в одном общем глобальном буфере вершин. Теперь мы должны пересоздать для каждого объекта буфер индексов, чтобы индексы корректно указывали на вершины в новом общем буфере вершин. Новые индексы вычисляются путем сложения старого индекса со значением смещения, указывающего с какой позиции в общем буфере вершин начинаются данные вершин объекта. Обратите внимание, что смещение измеряется в вершинах, а не в байтах. Direct3D позволяет передать значение смещения в параметре **BaseVertexIndex**, вместо того чтобы самостоятельно пересчитывать индексы в зависимости от того, в каком месте общего буфера вершин находится объект. Перерасчет индексов в этом случае Direct3D выполнит самостоятельно.

Рис. 3.2. *Объединение раздельно объявленных буферов вершин в один общий буфер вершин*

3.4.3 Начало и завершение сцены

И последний фрагмент информации: помните, что все вызовы методов рисования должны находиться внутри пары вызовов **`IDirect3DDevice9::BeginScene`** и **`IDirect3DDevice9::EndScene`**. К примеру, следует писать:

```
_device->BeginScene();
     _device->DrawPrimitive(...);
_device->EndScene();
```

3.5 Геометрические объекты D3DX

Создание трехмерных объектов путем прописывания в коде данных каждого, составляющего их треугольника, является утомительным занятием. К счастью, библиотека D3DX предлагает несколько методов, генерирующих данные сеток простых трехмерных объектов за нас.

Библиотека D3DX предоставляет следующие шесть методов для создания сеток:

- **`D3DXCreateBox`**
- **`D3DXCreateSphere`**
- **`D3DXCreateCylinder`**
- **`D3DXCreateTeapot`**
- **`D3DXCreatePolygon`**
- **`D3DXCreateTorus`**

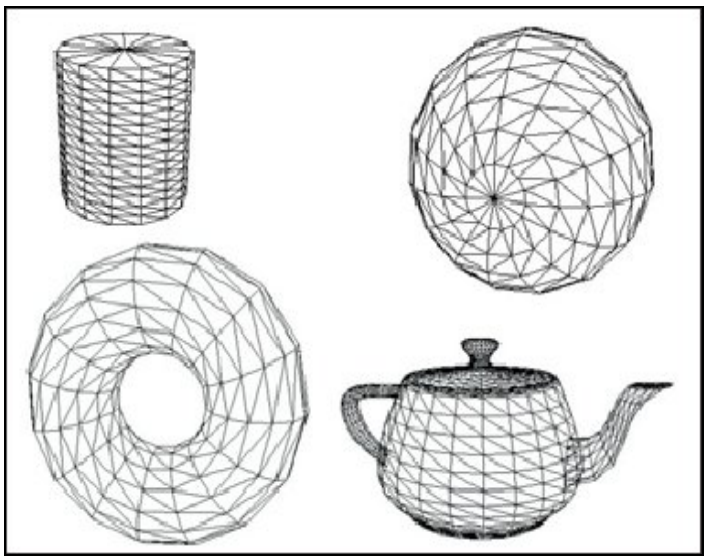

Рис. 3.3. *Объекты, создаваемые и визуализируемые с использованием функций* ***D3DXCreate****

Все шесть функций применяются одинаково и используют структуру данных сетки D3DX **ID3DXMesh** и интерфейс **ID3DXBuffer**. Подробнее об этих интерфейсах мы поговорим в главе 10 и главе 11. Сейчас мы проигнорируем детали их функционирования и сосредоточимся на простейшем варианте их использования.

```
HRESULT D3DXCreateTeapot(
     LPDIRECT3DDEVICE9 pDevice, // связанное с сеткой устройство
     LPD3DXMESH* ppMesh,        // указатель на полученную сетку
     LPD3DXBUFFER* ppAdjacency  // сейчас присвоить ноль
);
```

Вот пример использования функции **D3DXCreateTeapot**:

```
ID3DXMesh* mesh = 0;
D3DXCreateTeapot(_device, &mesh, 0);
```

После того, как мы сгенерировали данные сетки, можно нарисовать ее с помощью метода **ID3DXMesh::DrawSubset**. Единственный параметр этого метода идентифицирует подгруппу сетки. Генерируемые функциями **D3DXCreate*** сетки состоят из одной подгруппы, так что в этом параметре передается ноль. Вот пример визуализации сетки:

```
_device->BeginScene();
     mesh->DrawSubset(0);
_device->EndScene();
```

Когда вы завершите работу с сеткой, освободите ее:

```
_mesh->Release();
_mesh = 0;
```

3.6 Примеры приложений: треугольники, кубы, чайники, D3DXCreate*

В сопроводительных файлах, которые вы можете загрузить с веб-сайта этой книги, посвященный данной главе каталог содержит четыре приложения.

- Triangle — Это очень простое приложение, демонстрирующее как создать и визуализировать в каркасном режиме треугольник.
- Cube — Чуть более сложное, чем вариант с треугольником, данное приложение отображает визуализированный в каркасном режиме вращающийся куб.
- Teapot — Это приложение использует функцию **`D3DXCreateTeapot`**, чтобы создать и отобразить вращающийся чайник.
- D3DXCreate — Приложение создает и визуализирует несколько различных трехмерных объектов, которые можно создать с помощью функций **`D3DXCreate*`**.

Давайте кратко обсудим реализацию примера Cube. Остальные примеры вы можете изучить самостоятельно.

Приложение отображает куб, как показано на рис. 3.4. Проект и его полный исходный код находятся в сопроводительных файлах, которые можно загрузить с веб-сайта этой книги.

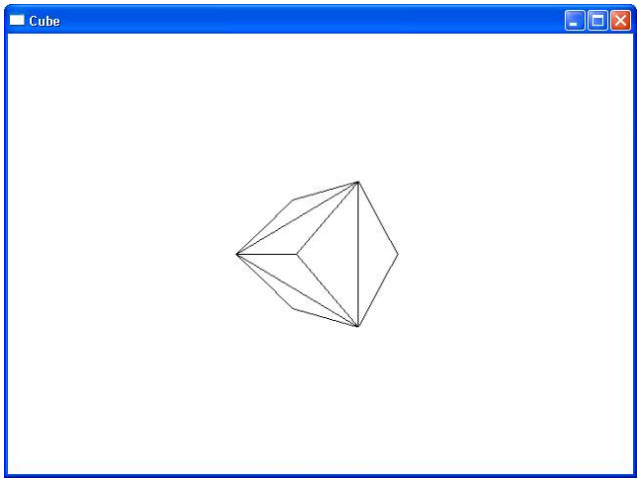

Рис. 3.4. *Окно приложения Cube*

Сперва мы объявляем две глобальных переменных, которые будут хранить данные вершин и индексов нашего куба:

```
IDirect3DVertexBuffer9* VB = 0;
IDirect3DIndexBuffer9*  IB = 0;
```

Кроме того, мы объявляем две глобальных константы, задающих разрешение экрана:

```
const int Width  = 800;
const int Height = 600;
```

Затем мы описываем структуру для хранения данных вершин и приводим описание настраиваемого формата вершин для этой структуры. В данном примере структура данных вершины содержит только информацию о местоположении вершины:

```
struct Vertex
{
     Vertex(){}
     Vertex(float x, float y, float z)
     {
          _x = x; _y = y; _z = z;
     }
     float _x, _y, _z;
     static const DWORD FVF;
};
const DWORD Vertex::FVF = D3DFVF_XYZ;
```

Давайте перейдем к функциям, образующим каркас приложения. Функция **Setup** создает буфер вершин и буфер индексов, блокирует их, записывает в буфер вершин данные образующих куб вершин, а в буфер индексов — индексы, описывающие образующие куб треугольники. Затем камера отодвигается на несколько единиц назад, чтобы мы могли увидеть куб, расположенный в начале мировой системы координат. После этого устанавливается преобразование проекции. В самом конце мы включаем каркасный режим визуализации:

```
bool Setup()
{
     // Создание буфера вершин и буфера индексов
     Device->CreateVertexBuffer(
                    8 * sizeof(Vertex),
                    D3DUSAGE_WRITEONLY,
                    Vertex::FVF,
                    D3DPOOL_MANAGED,
                    &VB,
                    0);

     Device->CreateIndexBuffer(
                    36 * sizeof(WORD),
                    D3DUSAGE_WRITEONLY,
                    D3DFMT_INDEX16,
                    D3DPOOL_MANAGED,
                    &IB,
                    0);

     // Заполнение буферов данными куба
     Vertex* vertices;
     VB->Lock(0, 0, (void**)&vertices, 0);
```

```
// Вершины единичного куба
vertices[0] = Vertex(-1.0f, -1.0f, -1.0f);
vertices[1] = Vertex(-1.0f,  1.0f, -1.0f);
vertices[2] = Vertex( 1.0f,  1.0f, -1.0f);
vertices[3] = Vertex( 1.0f, -1.0f, -1.0f);
vertices[4] = Vertex(-1.0f, -1.0f,  1.0f);
vertices[5] = Vertex(-1.0f,  1.0f,  1.0f);
vertices[6] = Vertex( 1.0f,  1.0f,  1.0f);
vertices[7] = Vertex( 1.0f, -1.0f,  1.0f);

VB->Unlock();

// Описание образующих куб треугольников
WORD* indices = 0;
IB->Lock(0, 0, (void**)&indices, 0);

// передняя грань
indices[0]  = 0; indices[1]  = 1; indices[2]  = 2;
indices[3]  = 0; indices[4]  = 2; indices[5]  = 3;

// задняя грань
indices[6]  = 4; indices[7]  = 6; indices[8]  = 5;
indices[9]  = 4; indices[10] = 7; indices[11] = 6;
// левая грань
indices[12] = 4; indices[13] = 5; indices[14] = 1;
indices[15] = 4; indices[16] = 1; indices[17] = 0;

// правая грань
indices[18] = 3; indices[19] = 2; indices[20] = 6;
indices[21] = 3; indices[22] = 6; indices[23] = 7;

// верх
indices[24] = 1; indices[25] = 5; indices[26] = 6;
indices[27] = 1; indices[28] = 6; indices[29] = 2;

// низ
indices[30] = 4; indices[31] = 0; indices[32] = 3;
indices[33] = 4; indices[34] = 3; indices[35] = 7;

IB->Unlock();

// размещение и ориентация камеры
D3DXVECTOR3 position(0.0f, 0.0f, -5.0f);
D3DXVECTOR3 target(0.0f, 0.0f, 0.0f);
D3DXVECTOR3 up(0.0f, 1.0f, 0.0f);
D3DXMATRIX V;
D3DXMatrixLookAtLH(&V, &position, &target, &up);
Device->SetTransform(D3DTS_VIEW, &V);

// установка матрицы проекции
D3DXMATRIX proj;
D3DXMatrixPerspectiveFovLH(
        &proj,
        D3DX_PI * 0.5f, // 90 градусов
```

```
            (float)Width / (float)Height,
            1.0f,
            1000.0f);
     Device->SetTransform(D3DTS_PROJECTION, &proj);

     // установка режима визуализации
     Device->SetRenderState(D3DRS_FILLMODE, D3DFILL_WIREFRAME);

     return true;
}
```

У метода **Display** две задачи: он должен обновлять сцену и затем визуализировать ее. Поскольку мы хотим, чтобы куб вращался, нам надо в каждом кадре увеличивать угол, определяющий насколько повернут куб. Из-за того, что угол в каждом кадре немного увеличивается, куб в каждом кадре чуть больше повернут, и в результате кажется, что он вращается. Обратите внимание, что для ориентации куба мы применяем мировое преобразование. Затем мы рисуем куб с помощью метода **IDirect3DDevice9::DrawIndexedPrimitive**.

```
bool Display(float timeDelta)
{
     if(Device)
     {
          //
          // Вращение куба:
          //
          D3DXMATRIX Rx, Ry;

          // поворот на 45 градусов вокруг оси X
          D3DXMatrixRotationX(&Rx, 3.14f / 4.0f);

          // увеличение угла поворота вокруг оси Y в каждом кадре
          static float y = 0.0f;
          D3DXMatrixRotationY(&Ry, y);
          y += timeDelta;

          // сброс угла поворота, если он достиг 2*PI
          if( y >= 6.28f )
               y = 0.0f;

          // комбинация поворотов
          D3DXMATRIX p = Rx * Ry;

          Device->SetTransform(D3DTS_WORLD, &p);

          //
          // Рисование сцены:
          //
          Device->Clear(0, 0,
                D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                0xffffffff, 1.0f, 0);
          Device->BeginScene();
```

```
            Device->SetStreamSource(0, VB, 0, sizeof(Vertex));
            Device->SetIndices(IB);
            Device->SetFVF(Vertex::FVF);
            Device->DrawIndexedPrimitive(D3DPT_TRIANGLELIST,
                                         0, 0, 8, 0, 12);

            Device->EndScene();
            Device->Present(0, 0, 0, 0);
        }
        return true;
}
```

Когда приложение завершает работу, мы освобождаем выделенную память. Это значит, что мы освобождаем интерфейсы буфера вершин и буфера индексов:

```
void Cleanup()
{
        d3d::Release<IDirect3DVertexBuffer9*>(VB);
        d3d::Release<IDirect3DIndexBuffer9*>(IB);
}
```

3.7 Итоги

- Данные вершин хранятся в интерфейсе **`IDirect3DVertexBuffer9`**. Аналогично, данные индексов хранятся в интерфейсе **`IDirect3DIndexBuffer9`**. Причина использования буферов вершин и индексов заключается в том, что их данные могут располагаться в видеопамяти.
- Статическая геометрия (то есть та, которая не изменяется в каждом кадре) должна храниться в статическом буфере вершин и индексов. В то же время динамическая геометрия (та, которая часто изменяется) должна храниться в динамическом буфере вершин и индексов.
- Режимы визуализации — это поддерживаемые устройством режимы, которые влияют на то как отображаются объекты. Режим визуализации оказывает влияние, пока не будет изменен, и установленное значение влияет на все последующие операции рисования. У всех режимов визуализации есть значение по умолчанию.
- Чтобы нарисовать содержимое буфера вершин и буфера индексов вы должны:
    - Вызвать метод **`IDirect3DDevice9::SetStreamSource`** и подключить к потоку тот буфер вершин, содержимое которого вы собираетесь рисовать.
    - Вызвать метод **`IDirect3DDevice9::SetFVF`** для указания формата визуализируемых вершин.

- Если вы используете буфер индексов, вызвать метод **`IDirect3DDevice9::SetIndices`** для указания используемого буфера индексов.
- Вызвать метод **`IDirect3DDevice9::DrawPrimitive`** либо **`IDirect3DDevice9::DrawIndexedPrimitive`** в коде расположенном между вызовами **`IDirect3DDevice9::BeginScene`** и **`IDirect3DDevice9::EndScene`**.

▪ Используя функции **`D3DXCreate*`** вы можете создавать достаточно сложные трехмерные объекты, такие как сферы, цилиндры и чайники.

Глава 4
Цвет

В предыдущей главе мы визуализировали объекты с помощью отрезков линий, так что они выглядели как проволочные каркасы. В этой главе мы узнаем, как можно раскрашивать визуализируемые объекты в разные цвета.

Цели

- Узнать, как в Direct3D задается цвет.
- Изучить способы закрашивания треугольников.

4.1 Представление цвета

В Direct3D цвета задаются путем указания тройки значений RGB. То есть мы указываем интенсивность красного, зеленого и синего цветов. Последующее смешивание этих трех компонентов дает в результате итоговый цвет. С помощью комбинаций красного, зеленого и синего мы можем представить миллионы цветов.

Для хранения информации о цвете мы будем использовать две различные структуры. Первая — это тип **D3DCOLOR**, который представляется значением **DWORD** и является 32-разрядным. Отдельные разряды в типе **D3DCOLOR** делятся на четыре 8-разрядных секции, каждая из которых хранит интенсивность отдельного компонента. Распределение значений показано на рис. 4.1.

Рис. 4.1. *32-разрядное представление цвета, где для каждого основного компоненнта (красного, зеленого и синего) выделено по одному байту. Четвертый байт отведен для альфа-канала*

Поскольку для каждого цвета выделено по байту памяти, его интенсивность может принимать значения от 0 до 255. Чем ближе значение к 0, тем меньше интенсивность цвета, и чем ближе значение к 255 — тем больше интенсивность.

ПРИМЕЧАНИЕ Сейчас не следует беспокоиться об альфа-составляющей, она используется для альфа-смешивания, о котором мы поговорим в главе 7.

Если задавать каждый компонент цвета и затем помещать его в требуемую позицию значения типа **D3DCOLOR**, потребуется выполнить несколько поразрядных операций. Direct3D предоставляет макрос с именем **D3DCOLOR_ARGB**, который выполнит все эти действия за нас. У макроса есть по одному параметру для каждой цветовой составляющей и один параметр для альфа-канала. Значение каждого параметра может быть от 0 до 255, а использование макроса выглядит так:

```
D3DCOLOR brightRed = D3DCOLOR_ARGB(255, 255, 0, 0);
D3DCOLOR someColor = D3DCOLOR_ARGB(255, 144, 87, 201);
```

Альтернативным способом является использование макроса **D3DCOLOR_XRGB**, который работает почти так же, но не получает значение для альфа-канала; вместо этого значение альфа-канала всегда устанавливается равным **0xff** (255).

```
#define D3DCOLOR_XRGB(r,g,b) D3DCOLOR_ARGB(0xff,r,g,b)
```

Вторым способом хранения данных цвета в Direct3D является структура **D3DCOLORVALUE**. В этой структуре для задания интенсивности каждого компонента цвета применяются значения с плавающей точкой. Доступный диапазон значений от 0 до 1; 0 соответствует отсутствию даного цвета, а 1 — его максимальной интенсивности.

```
typedef struct _D3DCOLORVALUE {
     float r;   // красная составляющая, диапазон 0.0-1.0
     float g;   // зеленая составляющая, диапазон 0.0-1.0
     float b;   // синяя составляющая, диапазон 0.0-1.0
     float a;   // альфа-составляющая, диапазон 0.0-1.0
} D3DCOLORVALUE;
```

Кроме того, мы можем использовать структуру **D3DXCOLOR**, которая содержит те же самые члены, что и **D3DCOLORVALUE**, но предоставляет дополнительные конструкторы и перегруженные операторы, упрощающие работу с цветами. А поскольку две структуры содержат одинаковые члены данных, можно выполнять преобразование типов из одного в другой и обратно. Определение **D3DXCOLOR** выглядит так:

```
typedef struct D3DXCOLOR
{
#ifdef __cplusplus
public:
     D3DXCOLOR() {}
     D3DXCOLOR(DWORD argb);
     D3DXCOLOR(CONST FLOAT *);
     D3DXCOLOR(CONST D3DXFLOAT16 *);
     D3DXCOLOR(CONST D3DCOLORVALUE&);
     D3DXCOLOR(FLOAT r, FLOAT g, FLOAT b, FLOAT a);

     // приведение типов
     operator DWORD () const;

     operator FLOAT* ();
     operator CONST FLOAT* () const;
     operator D3DCOLORVALUE* ();
     operator CONST D3DCOLORVALUE* () const;

     operator D3DCOLORVALUE& ();
     operator CONST D3DCOLORVALUE& () const;

     // операторы присваивания
     D3DXCOLOR& operator += (CONST D3DXCOLOR&);
     D3DXCOLOR& operator -= (CONST D3DXCOLOR&);
     D3DXCOLOR& operator *= (FLOAT);
     D3DXCOLOR& operator /= (FLOAT);

     // унарные операторы
     D3DXCOLOR operator + () const;
     D3DXCOLOR operator - () const;
```

```
     // бинарные операторы
     D3DXCOLOR operator + (CONST D3DXCOLOR&) const;
     D3DXCOLOR operator - (CONST D3DXCOLOR&) const;
     D3DXCOLOR operator * (FLOAT) const;
     D3DXCOLOR operator / (FLOAT) const;

     friend D3DXCOLOR operator * (FLOAT, CONST D3DXCOLOR&);

     BOOL operator == (CONST D3DXCOLOR&) const;
     BOOL operator != (CONST D3DXCOLOR&) const;

#endif //__cplusplus
     FLOAT r, g, b, a;
} D3DXCOLOR, *LPD3DXCOLOR;
```

ПРИМЕЧАНИЕ Обратите внимание, что в структурах **D3DCOLORVALUE** и **D3DXCOLOR** имеется по четыре значения с плавающей точкой. Это позволяет предствить цвет как четырехмерный вектор (*r*, *g*, *b*, *a*). Цветовые векторы могут складываться, вычитаться и масштабироваться как обычные. В то же время скалярное и векторное произведение цветовых векторов не имеют смысла. Однако, перемножение отдельных компонент для цветовых векторов имеет смысл. Так что оператор умножения цвета на цвет в классе **D3DXCOLOR** перемножает отдельные компоненты. Символ $\otimes$ означает перемножение отдельных компонентов, которое определяется следующим образом: $(c_1, c_2, c_3, c_4) \otimes (k_1, k_2, k_3, k_4) = (c_1k_1, c_2k_2, c_3k_3, c_4k_4)$

Мы дополним наш файл d3dUtility.h следующими глобальными цветовыми константами:

```
namespace d3d
{
   .
   .
   .
     const D3DXCOLOR     WHITE(D3DCOLOR_XRGB(255, 255, 255));
     const D3DXCOLOR     BLACK(D3DCOLOR_XRGB(  0,   0,   0));
     const D3DXCOLOR       RED(D3DCOLOR_XRGB(255,   0,   0));
     const D3DXCOLOR     GREEN(D3DCOLOR_XRGB(  0, 255,   0));
     const D3DXCOLOR      BLUE(D3DCOLOR_XRGB(  0,   0, 255));
     const D3DXCOLOR    YELLOW(D3DCOLOR_XRGB(255, 255,   0));
     const D3DXCOLOR      CYAN(D3DCOLOR_XRGB(  0, 255, 255));
     const D3DXCOLOR   MAGENTA(D3DCOLOR_XRGB(255,   0, 255));
}
```

4.2 Цвета вершин

Цвета примитивов задаются путем указания цветов образующих их вершин. Следовательно, необходимо добавить к структуре данных вершины члены для хранения цветовых компонентов. Обратите внимание, что здесь нельзя

использовать тип **D3DCOLORVALUE**, поскольку Direct3D ожидает описания цвета вершины в виде 32-разрядного значения. (Как ни странно, при работе с вершинными шейдерами мы должны использовать для задания цвета вершин четырехмерные цветовые векторы, и, следовательно, указывать 128-разрядное значение цвета, но сейчас не будем говорить об этом. Вершинные шейдеры рассматриваются в главе 17.)

```
struct ColorVertex
{
     float _x, _y, _z;
     D3DCOLOR _color;
     static const DWORD FVF;
}
const DWORD ColorVertex::FVF = D3DFVF_XYZ | D3DFVF_DIFFUSE;
```

4.3 Затенение

Затенение (shading) выполняется во время растеризации и определяет каким образом цвет вершин будет использоваться при вычислении цвета каждого из образующих примитив пикселей. Обычно используются два метода затенения: равномерное и затенение по методу Гуро.

При равномерном затенении все пиксели примитива окрашиваются в цвет, заданный для *первой* вершины примитива. Так что треугольник, образованный перечисленными ниже тремя вершинами, будет красным, поскольку его первая вершина — красная. Цвета второй и третьей вершин при равномерном затенении игнорируются.

```
ColorVertex t[3];
t[0]._color = D3DCOLOR_XRGB(255, 0, 0);
t[1]._color = D3DCOLOR_XRGB(0, 255, 0);
t[2]._color = D3DCOLOR_XRGB(0, 0, 255);
```

При равномерном затенении объекты выглядят угловатыми, поскольку нет плавных переходов от одного цвета к другому. Более качественным вариантом является затенение по алгоритму Гуро (также называемое гладким затенением). При затенении по методу Гуро цвета каждой точки определяются путем линейной интерполяции цветов вершин примитива. На рис. 4.2 показаны два треугольника: один закрашен с использованием равномерного затенения, а другой — с использованием затенения по методу Гуро.

Подобно многим другим вещам в Direct3D, режим затенения устанавливается через механизм режимов визуализации Direct3D.

```
// включение равномерной заливки
Device->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_FLAT);

// включение заливки по методу Гуро
Device->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_GOURAUD);
```

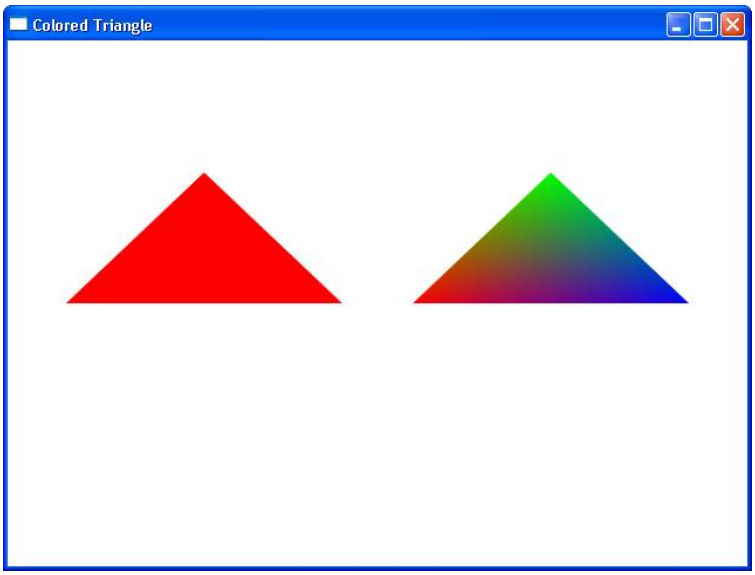

Рис. 4.2. *Слева равномерное затенение треугольника красным цветом. Справа треугольник с вершинами красного, зеленого и синего цвета, затенение которого выполнялось по методу Гуро; обратите внимание на интерполяцию цветов*

4.4 Пример приложения: цветные треугольники

Пример приложения из этой главы отображает закрашенные треугольники: для одного из них используется равномерное затенение, а для другого — затенение по методу Гуро. Результат визуализации показан на рис. 4.2. Сперва мы добавляем несколько глобальных переменных:

```
D3DXMATRIX World;
IDirect3DVertexBuffer9* Triangle = 0;
```

Матрица **`D3DXMATRIX`** будет использоваться для мирового преобразования тех треугольников, которые мы будем рисовать. Переменная **`Triangle`** является буфером вершин в котором хранятся вершины треугольника. Обратите внимание, что мы храним данные только одного треугольника и рисуем его несколько раз указывая различные матрицы мирового преобразования чтобы изменить его местоположение в сцене.

Метод **`Setup`** создает буфер вершин и заполняет его данными вершин, для каждой из которых также указывается цвет. Первая вершина треугольника становится красной, вторая — зеленой, а третья — синей. Потом для данного примера мы запрещаем освещение. Обратите внимание, что в данном примере используется новая структура данных вершин **`ColorVertex`**, которая была описана в разделе 4.2.

```
bool Setup()
{
     // Создание буфера вершин
     Device->CreateVertexBuffer(
                    3 * sizeof(ColorVertex),
                    D3DUSAGE_WRITEONLY,
                    ColorVertex::FVF,
                    D3DPOOL_MANAGED,
                    &Triangle,
                    0);

     // Заполнение буфера данными вершин треугольника
     ColorVertex* v;
     Triangle->Lock(0, 0, (void**)&v, 0);

     v[0] = ColorVertex(-1.0f, 0.0f, 2.0f,
                        D3DCOLOR_XRGB(255,   0,   0));
     v[1] = ColorVertex( 0.0f, 1.0f, 2.0f,
                        D3DCOLOR_XRGB(  0, 255,   0));
     v[2] = ColorVertex( 1.0f, 0.0f, 2.0f,
                        D3DCOLOR_XRGB(  0,   0, 255));

     Triangle->Unlock();

     // Установка матрицы проекции
     D3DXMATRIX proj;
     D3DXMatrixPerspectiveFovLH(
                    &proj,
                    D3DX_PI * 0.5f, // 90 градусов
                    (float)Width / (float)Height,
                    1.0f,
                    1000.0f);
     Device->SetTransform(D3DTS_PROJECTION, &proj);

     // Установка режима визуализации
     Device->SetRenderState(D3DRS_LIGHTING, false);

     return true;
}
```

Затем функция **Display** дважды рисует объект **Triangle** в двух различных местах и с различными режимами затенения. Позиция каждого треугольника задается матрицей мирового преобразования **World**.

```
bool Display(float timeDelta)
{
    if(Device)
    {
        Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                      0xffffffff, 1.0f, 0);
        Device->BeginScene();

        Device->SetFVF(ColorVertex::FVF);
        Device->SetStreamSource(0, Triangle, 0, sizeof(ColorVertex));
```

```
        // Рисуем левый треугольник с равномерной заливкой
        D3DXMatrixTranslation(&World, -1.25f, 0.0f, 0.0f);
        Device->SetTransform(D3DTS_WORLD, &World);

        Device->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_FLAT);
        Device->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);

        // Рисуем правый треугольник с заливкой Гуро
        D3DXMatrixTranslation(&World, 1.25f, 0.0f, 0.0f);
        Device->SetTransform(D3DTS_WORLD, &World);

        Device->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_GOURAUD);
        Device->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);

        Device->EndScene();
        Device->Present(0, 0, 0, 0);
    }
    return true;
}
```

4.5 Итоги

- Цвета задаются путем указания их красной, зеленой и синей составляющих. Смешивание трех базовых цветов различной интенсивности позволяет задавать миллионы различных цветов. В Direct3D для опиисания цветов в коде могут использоваться типы **D3DCOLOR**, **D3DCOLORVALUE** или **D3DXCOLOR**.
- Иногда мы будем рассматривать цвет как четырехмерный вектор (r, g, b, a). Цветовые векторы могут складываться, вычитаться и масштабироваться как обычные. В то же время операции скалярного или векторного произведения для цветовых векторов не имеют смысла. Имеет смысл операция перемножения компонент двух цветовых векторов, которая обозначается знаком $\otimes$ и определена следующим образом: $(c_1, c_2, c_3, c_4) \otimes (k_1, k_2, k_3, k_4) = (c_1k_1, c_2k_2, c_3k_3, c_4k_4)$.
- Мы задаем цвета вершин, а Direct3D во время растеризации определяет цвет каждого пиксела грани с учетом выбранного режима затенения.
- При равномерном затенении все пикселы примитива окрашиваются в цвет его первой вершины. При затенении по методу Гуро цвет каждого пикселя примитива вычисляется путем линейной интерполяции значений цветов его вершин.

Глава 5
Освещение

Чтобы добавить нашим сценам реализма необходимо освещение. Кроме того, освещение позволяет подчеркивать форму и объем объектов. Если мы используем освещение, нам больше не надо самостоятельно задавать цвета вершин; Direct3D обработает каждую вершину в своем механизме расчета освещенности и вычислит ее цвет основываясь на данных об источниках света, материале и ориентации поверхности относительно источников света. Вычисление цвета вершины на основе освещения модели дает более естественные результаты.

Цели

- Изучить источники освещения, поддерживаемые Direct3D, и типы освещения, которое эти источники могут давать.
- Разобраться, как задать взаимодействие света с поверхностью на которую он падает.
- Узнать, как математически задается ориентация треугольной грани, чтобы вычислить под каким углом на нее падает свет.

5.1 Компоненты света

В модели освещения Direct3D свет, испускаемый источниками, состоит из трех составляющих, или трех типов света:

- **Фоновый свет** (ambient light) — Этот тип освещения моделирует свет, который отражается от других поверхностей и освещает всю сцену. Например, в сцене часто освещены те части объектов, которые не находятся в прямой видимости источника света. Эти части освещаются тем светом,который отражается от других поверхностей. Фоновый свет — это трюк, который используется для приблизительного моделирования этого отраженного света.
- **Рассеиваемый свет** (diffuse light) — Этот свет распространяется в заданном направлении. Сталкиваясь с поверхностью он отражается равномерно во всех направлениях. Поэтому интенсивность достигшего глаза зрителя света не зависит от точки, с которой просматривается сцена, и местоположение зрителя можно не учитывать. Следовательно, при вычислении расеянного освещения надо учитывать только направление световых лучей и позицию поверхности. Это основная составляющая испускаемого источником света.
- **Отражаемый свет** (specular light) — Этот свет распространяется в заданном направлении. Сталкиваясь с поверхностью он отражается строго в одном направлении, формируя блики, которые видимы только при взгляде на объект под определенным углом. Поскольку свет отражается только в одном направлении, ясно что при расчете отражаемого света необходимо принимать во внимание местоположение и ориентацию камеры, направление световых лучей и ориентацию поверхности. Отражаемый свет используется для моделирования света, формируемого освещенными объектами, такого как блики, появляющиеся при освещении полированных поверхностей.

Отражаемый свет требует гораздо большего объема вычислений, чем другие составляющие освещения; поэтому Direct3D позволяет отключать его. Фактически, по умолчанию эта составляющая не используется; чтобы включить ее, необходимо установить режим визуализации **D3DRS_SPECULARENABLE**.

```
Device->SetRenderState(D3DRS_SPECULARENABLE, true);
```

Каждая составляющая освещения представляется структурой **D3DCOLORVALUE** или **D3DXCOLOR**, которая определяет цвет. Вот несколько примеров разноцветных источников света:

```
D3DXCOLOR redAmbient(1.0f, 0.0f, 0.0f, 1.0f);
D3DXCOLOR blueDiffuse(0.0f, 0.0f, 1.0f, 1.0f);
D3DXCOLOR whiteSpecular(1.0f, 1.0f, 1.0f, 1.0f);
```

ПРИМЕЧАНИЕ Когда структура **D3DXCOLOR** используется для описания источника света, альфа-компонента игнорируется.

5.2 Материалы

Цвет объектов, которые мы видим в реальном мире, определяется цветом отражаемого ими света. Например, красный шар выглядит красным потому что он поглощает все лучи света, кроме красных. Красный свет отражается от шара и попадает в глаз зрителя, и мы считаем, что шар красный. В Direct3D данное явление моделируется путем задания материала объекта. Материал позволяет задать отражающую способность поверхности. В коде материал представляется с помощью структуры **D3DMATERIAL9**.

```
typedef struct _D3DMATERIAL9 {
     D3DCOLORVALUE Diffuse, Ambient, Specular, Emissive;
     float Power;
} D3DMATERIAL9;
```

- **Diffuse** — Задает количество отражаемого поверхностью рассеиваемого света.
- **Ambient** — Задает количество отражаемого поверхностью фонового света.
- **Specular** — Задает количество отражаемого поверхностью отражаемого света.
- **Emissive** — Данный компонент позволяет увеличивать значения цветов, что создает эффект свечения поверхности.
- **Power** — Задает резкость зеркальных отражений; чем больше значение, тем более резкими будут отражения.

Предположим. для примера, что мы хотим создать красный шар. Мы должны определить материал шара таким образом, чтобы он отражал только красную составляющую света, а все остальные поглощал:

```
D3DMATERIAL9 red;
::ZeroMemory(&red, sizeof(red));
red.Diffuse  = D3DXCOLOR(1.0f, 0.0f, 0.0f, 1.0f); // красный
red.Ambient  = D3DXCOLOR(1.0f, 0.0f, 0.0f, 1.0f); // красный
red.Specular = D3DXCOLOR(1.0f, 0.0f, 0.0f, 1.0f); // красный
red.Emissive = D3DXCOLOR(0.0f, 0.0f, 0.0f, 1.0f); // нет свечения
red.Power = 5.0f;
```

Здесь мы присваиваем зеленой и синей составляющим 0, указывая, что материал полностью поглощает лучи света данных цветов. Красной составляющей мы присваиваем значение 1, а это значит, что лучи данного цвета полностью отражаются поверхностью. Обратите внимание,что мы можем контроллировать степень отражения для каждой составляющей освещения (фоновой, рассеиваемой и отражаемой).

Также обратите внимание, что если мы создаем источник света испускающий лучи только синего цвета, то результат освещения данного шара может нас разочаровать, поскольку шар полностью поглощает лучи синего цвета а лучи красного цвета на него не попадают. Если объект поглощает все падающие на

него лучи света, то он выглядит черным. Точно так же если объект полностью отражает все лучи (красный, синий и зеленый), то он выглядит белым. Поскольку ручное задание параметров материалов является скучным занятием, мы добавим в файлы d3dUtility.h/cpp несколько вспомогательных функций и глобальных констант материалов:

```
D3DMATERIAL9 d3d::InitMtrl(D3DXCOLOR a, D3DXCOLOR d,
                           D3DXCOLOR s, D3DXCOLOR e, float p)
{
     D3DMATERIAL9 mtrl;
     mtrl.Ambient  = a;
     mtrl.Diffuse  = d;
     mtrl.Specular = s;
     mtrl.Emissive = e;
     mtrl.Power    = p;
     return mtrl;
}

namespace d3d
{
.
.
.
D3DMATERIAL9 InitMtrl(D3DXCOLOR a, D3DXCOLOR d,
                      D3DXCOLOR s, D3DXCOLOR e, float p);

const D3DMATERIAL9 WHITE_MTRL  = InitMtrl(WHITE, WHITE,
                                          WHITE, BLACK, 8.0f);

const D3DMATERIAL9 RED_MTRL    = InitMtrl(RED, RED,
                                          RED, BLACK, 8.0f);

const D3DMATERIAL9 GREEN_MTRL  = InitMtrl(GREEN, GREEN,
                                          GREEN, BLACK, 8.0f);

const D3DMATERIAL9 BLUE_MTRL   = InitMtrl(BLUE, BLUE,
                                          BLUE, BLACK, 8.0f);

const D3DMATERIAL9 YELLOW_MTRL = InitMtrl(YELLOW, YELLOW,
                                          YELLOW, BLACK, 8.0f);
}
```

ПРИМЕЧАНИЕ По адресу *http://www.adobe.com/support/techguides/color/color-theory/main.html* находится замечательная статья, посвященная теории цвета, освещению и тому, как человеческий глаз различает цвета.

В структуре данных вершины нет членов данных для задания свойств материала; вместо этого нам надо задать используемый материал с помощью метода **`IDirect3DDevice9::SetMaterial(CONST D3DMATERIAL9* pMaterial)`**.

Если мы хотим визуализировать несколько объектов, используя различные материалы, нам надо написать следующее:

```
D3DMATERIAL9 blueMaterial, redMaterial;

...// инициализация структур материалов

Device->SetMaterial(&blueMaterial);
drawSphere(); // синяя сфера

Device->SetMaterial(&redMaterial);
drawSphere(); // красная сфера
```

5.3 Нормали вершин

Нормалью грани (*face normal*) называется вектор, определяющиий ориентацию лицевой грани многоугольника (рис. 5.1).

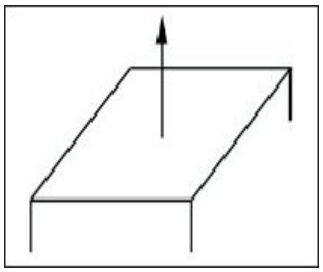

Рис. 5.1. *Лицевая нормаль поверхности*

Нормали вершин (*vertex normals*) основаны на той же самой идее, но в этом случае задается не нормаль для всего многоугольника, а отдельная нормаль для каждой образующей его вершины (рис. 5.2).

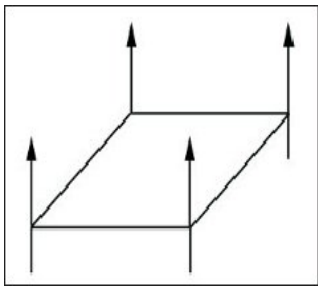

Рис. 5.2. *Нормали вершин поверхности*

Direct3D необходимо знать нормали вершин, поскольку они необходимы чтобы оперделить под каким углом свет падает на грань. Кроме того, поскольку вычисление освещенности выполняется для каждой из вершин, Direct3D необходимо знать ориентацию грани (нормаль) для каждой вершины. Обратите внимание, что нормаль вершины не всегда совпадает с нормалью грани. Наиболее распростарненным примером объекта у которого нормали треугольных граней не совпадают с нормалями вершин является сфера или цилиндр (рис. 5.3).

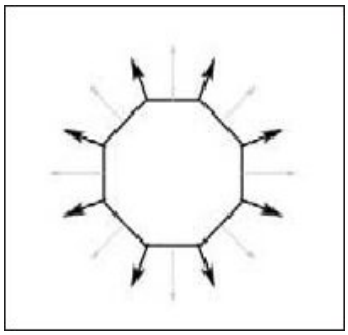

Рис. 5.3. *Пример объекта, у которого нормали вершин не совпадают с нормалями граней. Векторы нормалей вершин выделены черным цветом, а векторы нормалей граней — серым*

Для описания нормалей вершин нам необходимо добавить соответствующие члены в структуру данных вершины:

```
struct Vertex
{
     float _x, _y, _z;
     float _nx, _ny, _nz;
     static const DWORD FVF;
}
const DWORD Vertex::FVF = D3DFVF_XYZ | D3DFVF_NORMAL;
```

Обратите внимание, что мы убрали члены данных, задающие цвет вершины, которые использовали в предыдущей главе. Дело в том, что теперь для вычисления цвета вершин мы будем использовать данные освещения.

Для простых объектов, таких как кубы и сферы, нормали вершин можно оперделить путем осмотра. Для сложных сеток необходим более алгоритмизированный способ. Предположим, что треугольник образован вершинами $\mathbf{p}_0$, $\mathbf{p}_1$ и $\mathbf{p}_2$, и нам необходимо вычислить нормали $\mathbf{n}_0$, $\mathbf{n}_1$ и $\mathbf{n}_2$ для каждой из вершин.

Простейший подход заключается в том, чтобы вычислить нормаль грани для треугольника и использовать ее в качестве нормали для всех трех вершин. Сперва вычислим два вектора, лежащих в полскости треугольника:

$$\mathbf{p}_1 - \mathbf{p}_0 = \mathbf{u}$$

$$\mathbf{p}_2 - \mathbf{p}_0 = \mathbf{v}$$

Тогда нормаль грани вычисляется по формуле:

$$\mathbf{n} = \mathbf{u} \times \mathbf{v}$$

Поскольку нормаль каждой вершины совпадает с нормалью грани:

$$\mathbf{n}_0 = \mathbf{n}_1 = \mathbf{n}_2 = \mathbf{n}$$

Ниже приведена функция, которая вычисляет нормаль треугольной грани на основании координат трех ее вершин. Обратите внимание, что функция предполагает, что вершины перечислены по часовой стрелке. Если это не так, нормаль будет указывать в противоположном направлении.

```
void ComputeNormal(D3DXVECTOR3* p0,
                   D3DXVECTOR3* p1,
                   D3DXVECTOR3* p2,
                   D3DXVECTOR3* out)
{
     D3DXVECTOR3 u = *p1 - *p0;
     D3DXVECTOR3 v = *p2 - *p0;

     D3DXVec3Cross(out, &u, &v);
     D3DXVec3Normalize(out, out);
}
```

Использование нормали грани в качестве нормалей вершин не позволяет добиться гладкого изображения состоящих из треугольных граней сложных кривых поверхностей. Лучшим методом вычисления нормалей вершин является *усреднение нормалей* (*normal averaging*). Чтобы вычислить вектор нормали $\mathbf{v}_n$ для вершины $\mathbf{v}$, мы вычисляем нормали граней всех треугольников сетки, в которые входит данная вершина $\mathbf{v}$. Затем вектор нормали вершины $\mathbf{v}_n$ получается путем вычисления среднего значения всех этих нормалей граней. Давайте разберем конкретный пример. Предположим, вершина $\mathbf{v}$ входит в три треугольника, для которых известны их нормали граней $\mathbf{n}_0$, $\mathbf{n}_1$ и $\mathbf{n}_2$. Тогда $\mathbf{v}_n$ вычисляется путем усреднения нормалей граней:

$$\mathbf{v}_n = \frac{1}{3}(\mathbf{n}_0 + \mathbf{n}_1 + \mathbf{n}_2)$$

В процессе преобразований может получиться так, что векторы нормалей станут денормализованными. Так что лучше всего предусмотреть возможность подобной ситуации и приказать Direct3D заново нормализовать все векторы нормалей после преобразований, включив режим визуализации **`D3DRS_NORMALIZENORMALS`**:

```
Device->SetRenderState(D3DRS_NORMALIZENORMALS, true);
```

5.4 Источники света

Direct3D поддерживает источники света трех типов.

- **Точечный свет** (point light) — У этого источника света есть местоположение в пространстве и он испускает свет во всех направлениях.

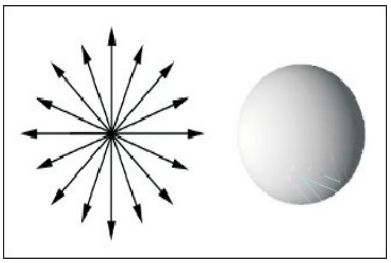

Рис. 5.4. *Точечный свет*

- **Направленный свет** (directional light) — У этого источника света нет местоположения, он испускает параллельные световые лучи в заданном направлении.

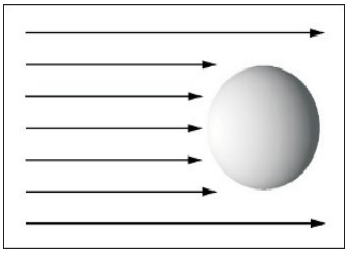

Рис. 5.5. *Направленный свет*

- **Зональный свет** (spot light) — Источник света данного типа похож на фонарик; у него есть местоположение и он испускает конический сноп лучей в заданном направлении. Световой конус характеризуется двумя углами — φ и θ. Угол φ задает размер внутреннего конуса, а угол θ — внешнего.

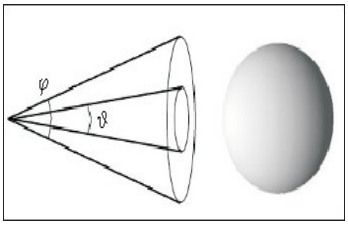

Рис. 5.6. *Зональный свет*

В коде источники света представляются структурой **D3DLIGHT9**.

```
typedef struct _D3DLIGHT9 {
     D3DLIGHTTYPE Type;
     D3DCOLORVALUE Diffuse;
     D3DCOLORVALUE Specular;
     D3DCOLORVALUE Ambient;
     D3DVECTOR Position;
     D3DVECTOR Direction;
     float Range;
     float Falloff;
     float Attenuation0;
     float Attenuation1;
     float Attenuation2;
     float Theta;
     float Phi;
} D3DLIGHT9;
```

- **Type** — Задает тип источника света и может принимать одно из трех значений: **D3DLIGHT_POINT**, **D3DLIGHT_SPOT** или **D3DLIGHT_DIRECTIONAL**.
- **Diffuse** — Цвет рассеиваемой составляющей испускаемого источником света.

- **Specular** — Цвет отражаемой составляющей испускаемого источником света.
- **Ambient** — Цвет фоновой составляющей испускаемого источником света.
- **Position** — Вектор, задающий местоположение источника света в пространстве. Для направленного света значение не используется.
- **Direction** — Вектор, задающий направление, в котором распространяется свет. Для точечного света не используется.
- **Range** — Максимальное расстояние, на которое может распространиться свет прежде чем окончательно потухнет. Значение не может быть больше чем $\sqrt{FLT\_MAX}$ и не оказывает влияния на направленный свет.
- **Falloff** — Значение используется только для зонального света. Оно определяет как меняется интенсивность света в пространстве между внутренним и внешним конусами. Обычно этому параметру присваивают значение 1.0f.
- **Attenuation0**, **Attenuation1**, **Attenuation2** — Переменные затухания, определяющие как меняется интенсивность света с увеличением расстояния до источника света. Эти переменные используются только для точечного и зонального света. Переменная **Attenuation0** задает постоянное затухание, **Attenuation1** — линейное затухание и **Attenuation2** — квадратичное затухание. Вычисления выполняются по формуле

$$Attenuation = \frac{1}{A_0 + A_1 \cdot D + A_2 \cdot D^2}$$

  где D — это расстояние от источника света, а A_0, A_1, A_2 соответственно **Attenuation0**, **Attenuation1** и **Attenuation2**.
- **Theta** — Используется только для зонального света; задает угол внутреннего конуса в радианах.
- **Phi** — Используется только для зонального света; задает угол внешнего конуса в радианах.

Подобно инициализации структуры **D3DMATERIAL9**, в том случае, когда нам нужны только простые источники света, инициализация структуры **D3DLIGHT9** становится рутинным занятием. Поэтому для инициализации простых источников света мы добавим в файлы d3dUtility.h/cpp следующие функции:

```
namespace d3d
{
.
.
.

D3DLIGHT9 InitDirectionalLight(D3DXVECTOR3* direction,
                               D3DXCOLOR* color);
```

```
D3DLIGHT9 InitPointLight(D3DXVECTOR3* position,
                         D3DXCOLOR* color);

D3DLIGHT9 InitSpotLight(D3DXVECTOR3* position,
                        D3DXVECTOR3* direction,
                        D3DXCOLOR* color);
}
```

Реализация этих функций не содержит сложных моментов. Мы рассмотрим только реализацию **`InitDirectionalLight`**. Остальные функции похожи на нее:

```
D3DLIGHT9 d3d::InitDirectionalLight(D3DXVECTOR3* direction,
                                    D3DXCOLOR* color)
{
     D3DLIGHT9 light;
     ::ZeroMemory(&light, sizeof(light));

     light.Type      = D3DLIGHT_DIRECTIONAL;
     light.Ambient   = *color * 0.4f;
     light.Diffuse   = *color;
     light.Specular  = *color * 0.6f;
     light.Direction = *direction;

     return light;
}
```

Теперь для создания источника направленного света белого цвета, испускающего лучи вдоль оси X в положительном направлении, можно написать:

```
D3DXVECTOR3 dir(1.0f, 0.0f, 0.0f);
D3DXCOLOR   c = d3d::WHITE;
D3DLIGHT9 dirLight = d3d::InitDirectionalLight(&dir, &c);
```

После того, как мы инициализировали экземпляр **`D3DLIGHT9`**, нам надо зарегистрировать его во внутреннем списке управляемых Direct3D источников света. Делается это вот так:

```
Device->SetLight(
     0,       // устанавливаемый элемент списка источников света,
              // диапазон 0 – maxlights
     &light); // адрес инициализированной структуры D3DLIGHT9
```

После регистрации источника света мы можем включать его и выключать, как показано в приведенном ниже фрагменте кода:

```
Device->LightEnable(
     0,     // Включаемый или выключаемый источник света в списке
     true); // true = включить, false = выключить
```

5.5 Пример приложения: освещенная пирамида

Пример приложения из этой главы создает сцену, показанную на рис. 5.7. Он показывает как задать нормали вершин, как создать материал и как создать и активизировать источник направленного света. Обратите внимание, что в этом примере мы не пользуемся функциями и объявлениями для материала и источника света из файлов d3dUtility.h/cpp потому что хотим сперва показать вам как можно всю работу выполнить вручную. Однако в остальных примерах из этой книги мы будем пользоваться вспомогательными функциями для задания материалов и источников света.

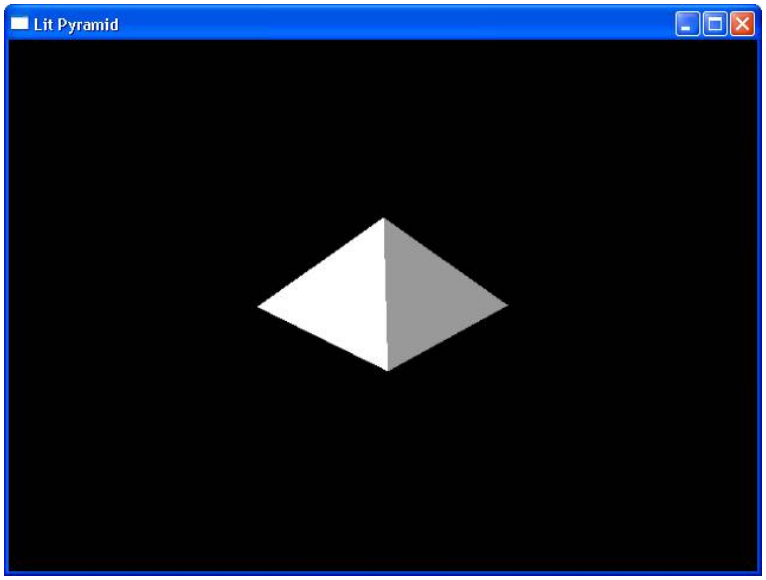

Рис. 5.7. *Окно приложения LitPyramid*

Вот действия, которые необходимо выполнить, чтобы добавить к сцене источник света:

1. Разрешить освещение.
2. Создать материал для каждого объекта и задать используемый материал перед визуализацией соответствующего объекта.
3. Создать один или несколько источников света, расставить их на сцене и включить их.
4. Если необходимо, включить дополнительные режимы освещения, например, обработку отражаемого света.

Сперва мы объявляем глобальный буфер вершин в котором будут храниться данные вершин пирамиды:

```
IDirect3DVertexBuffer9* Pyramid = 0;
```

Весь код, относящийся к рассматриваемой в данной главе теме, находится в функции **Setup**, поэтому для экономии места остальные функции мы рассматривать не будем. В функции **Setup** реализованы перечисленные выше этапы добавления к сцене освещения. Начинается функция с разрешения использования освещения, которое не является необходимым, поскольку по умолчанию освещение разрешено (но хуже от этого не будет).

```
bool Setup()
{
     Device->SetRenderState(D3DRS_LIGHTING, true);
```

Затем мы создаем буфер вершин, блокируем его и задаем данные вершин, образующих треугольную пирамиду. Нормали вершин вычисляются с помощью описанного в разделе 5.3 алгоритма. Обратите внимание, что хотя треугольники совместно используют некоторые вершины, нормали у каждого из них свои; так что использование для данного объекта списка индексов не принесет никакой пользы. Например, все грани используют вершину (0, 1, 0), находящуюся на верху пирамиды; однако нормаль этой вершины у каждого треугольника указывает в своем направлении.

```
     Device->CreateVertexBuffer(
               12 * sizeof(Vertex),
               D3DUSAGE_WRITEONLY,
               Vertex::FVF,
               D3DPOOL_MANAGED,
               &Pyramid,
               0);

     // Заполняем буфер вершин данными пирамиды
     Vertex* v;
     Pyramid->Lock(0, 0, (void**)&v, 0);

     // передняя грань
     v[0] = Vertex(-1.0f, 0.0f, -1.0f, 0.0f, 0.707f, -0.707f);
     v[1] = Vertex( 0.0f, 1.0f,  0.0f, 0.0f, 0.707f, -0.707f);
     v[2] = Vertex( 1.0f, 0.0f, -1.0f, 0.0f, 0.707f, -0.707f);

     // левая грань
     v[3] = Vertex(-1.0f, 0.0f,  1.0f, -0.707f, 0.707f, 0.0f);
     v[4] = Vertex( 0.0f, 1.0f,  0.0f, -0.707f, 0.707f, 0.0f);
     v[5] = Vertex(-1.0f, 0.0f, -1.0f, -0.707f, 0.707f, 0.0f);

     // правая грань
     v[6] = Vertex( 1.0f, 0.0f, -1.0f, 0.707f, 0.707f, 0.0f);
     v[7] = Vertex( 0.0f, 1.0f,  0.0f, 0.707f, 0.707f, 0.0f);
     v[8] = Vertex( 1.0f, 0.0f,  1.0f, 0.707f, 0.707f, 0.0f);

     // задняя грань
     v[9]  = Vertex( 1.0f, 0.0f, 1.0f, 0.0f, 0.707f, 0.707f);
     v[10] = Vertex( 0.0f, 1.0f, 0.0f, 0.0f, 0.707f, 0.707f);
     v[11] = Vertex(-1.0f, 0.0f, 1.0f, 0.0f, 0.707f, 0.707f);

     Pyramid->Unlock();
```

После того, как данные вершин для нашего объекта сгенерированы, мы описываем взаимодействие объекта со световыми лучами путем задания материала. В данном примере пирамида отражает белый свет, сама не испускает света и формирует блики.

```
    D3DMATERIAL9 mtrl;
         mtrl.Ambient  = d3d::WHITE;
         mtrl.Diffuse  = d3d::WHITE;
         mtrl.Specular = d3d::WHITE;
         mtrl.Emissive = d3d::BLACK;
         mtrl.Power    = 5.0f;

    Device->SetMaterial(&mtrl);
```

Теперь мы создаем и включаем источник направленного света. Лучи направленного света распространяются параллельно оси X в положительном направлении. Рассеиваемая составляющая света окрашена в белый цвет и имеет максимальную интенсивность (**`dir.Diffuse = WHITE`**), отражаемая составляющая также белого цвета, но малой интенсивности (**`dir.Specular = WHITE * 0.3f`**), а фоновая составляющая — белого цвета и средней интенсивности (**`dir.Ambient = WHITE * 0.6f`**).

```
    D3DLIGHT9 dir;
    ::ZeroMemory(&dir, sizeof(dir));

    dir.Type      = D3DLIGHT_DIRECTIONAL;
    dir.Diffuse   = d3d::WHITE;
    dir.Specular  = d3d::WHITE * 0.3f;
    dir.Ambient   = d3d::WHITE * 0.6f;
    dir.Direction = D3DXVECTOR3(1.0f, 0.0f, 0.0f);

    Device->SetLight(0, &dir);
    Device->LightEnable(0, true);
```

И, наконец, мы устанавливаем режимы визуализации для ренормализации нормалей и разрешения обработки отражаемой составляющей света.

```
    Device->SetRenderState(D3DRS_NORMALIZENORMALS, true);
    Device->SetRenderState(D3DRS_SPECULARENABLE, true);

    // ... код инициализации матрицы вида и матрицы проекции опущен

    return true;
}
```

5.6 Дополнительные примеры

В сопроводительные файлы к этой главе включены три дополнительных примера. Для создания образующих сцену трехмерных объектов в них используются функции **`D3DXCreate*`**. Функции **`D3DXCreate*`** создают данные вершин в

формате **D3DFVF_XYZ | D3DFVF_NORMAL**. Кроме того, эти функции за нас вычисляют нормали вершин для каждой сетки. Дополнительные примеры демонстрируют как использовать направленный, точечный и зональный свет. На рис. 5.8 показано окно программы, демонстрирующей использование направленного света.

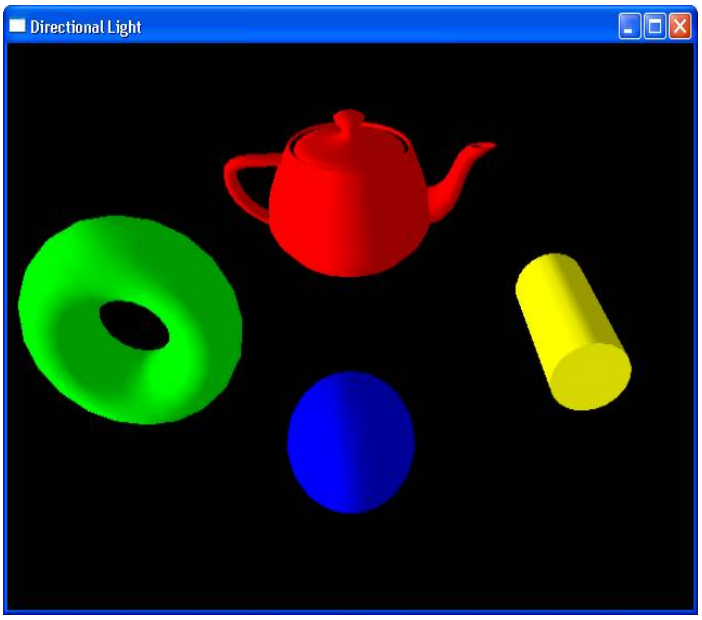

Рис. 5.8. *Окно программы DirectionalLight*

5.7 Итоги

- Direct3D поддерживает три модели источников света: направленный свет, точечный свет и зональный свет. Испускаемый источником свет состоит из трех компонентов: фоновый свет, рассеиваемый свет и отражаемый свет.
- Материал поверхности описывает взаимодействие поверхности с падающими на нее лучами света (то есть, сколько света поверхность отражает, а сколько поглощает, что определяет цвет поверхности).
- Нормали вершин используются для задания ориентации вершин. Они применяются для того. чтобы механизм визуализации Direct3D мог определить под каким углом лучи света падают на вершину. Иногда нормали вершин совпадают с нормалью образуемой ими треугольной грани, но для представления гладких объектов (таких как сферы и цилиндры) такой подход не годится.

Глава 6
Текстурирование

Наложение текстур (*texture mapping*) — это техника, позволяющая наложить изображение на треугольную грань; ее применение во много раз увеличивает детализованность и реализм сцены. Например, мы можем создать куб и превратить его в деревянный ящик, наложив на каждую грань текстуру с изображением стенки ящика (рис. 6.1).

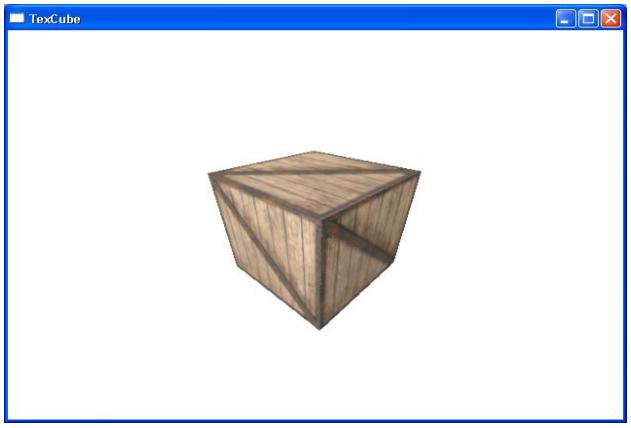

Рис. 6.1. *Куб с текстурой деревянного ящика*

В Direct3D текстура представляется с помощью интерфейса **`IDirect3DTexture9`**. Текстура, подобно поверхности, является двухмерной матрицей пикселей и может быть наложена на треугольную грань.

Цели

- Узнать, как задать фрагмент текстуры, который будет наложен на треугольную грань.
- Изучить способы создания текстур.
- Исследовать применение фильтрации текстур для сглаживания получающегося изображения.

6.1 Координаты текстур

Direct3D использует для текстур систему координат, образованную горизонтальной осью U и вертикальной осью V. Пара координат (u, v) идентифицирует элемент текстуры, называемый *текселем* (*texel*). Обратите внимание, что ось V направлена вниз (рис. 6.2).

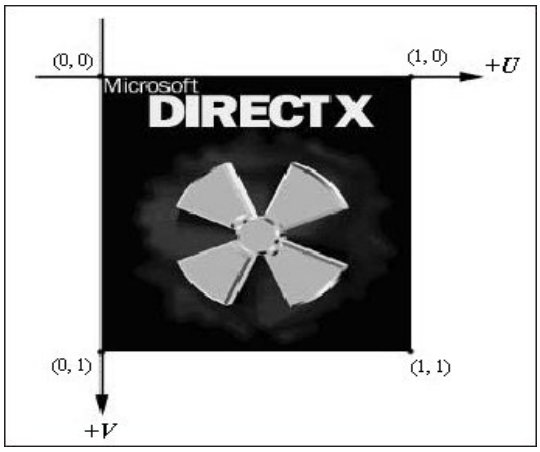

Рис. 6.2. *Система координат текстуры иногда называемая пространством текстуры*

Кроме того, обратите внимание, на нормализованные координаты в диапазоне [0, 1], которые обеспечивают для Direct3D работу со значениями из фиксированного диапазона, не зависящего от размеров конкретной текстуры.

Для каждой треугольной грани в трехмерном пространстве мы должны определить соответствующий треугольный фрагмент текстуры, который будет наложен на эту грань (рис. 6.3).

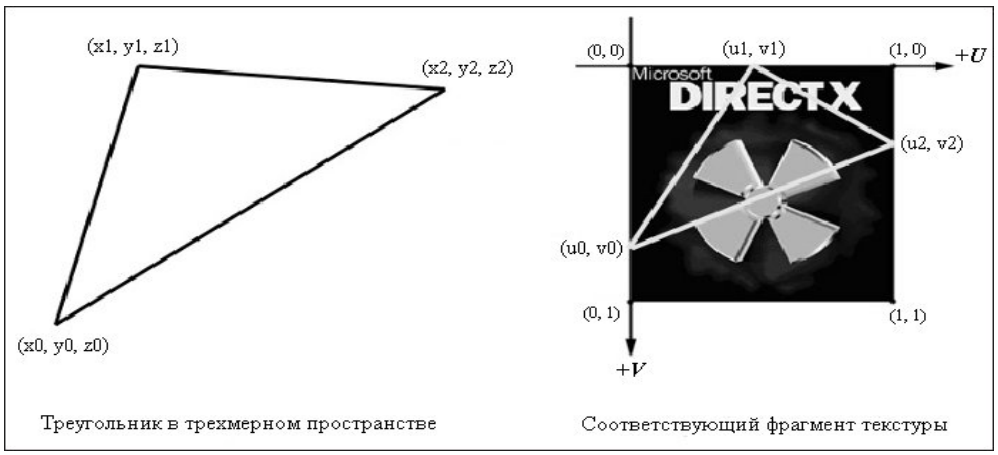

Рис. 6.3. *Слева изображена треугольная грань в трехмерном пространстве, а справа — двухмерный треугольный фрагмент текстуры, который должен быть наложен на данную грань*

Для этого мы еще раз модифицируем нашу структуру данных вершины и добавим в нее пару координат текстуры, которые будут определять соответствие между вершиной и точкой текстуры.

```
struct Vertex
{
     float _x, _y, _z;
     float _nx, _ny, _nz;
     float _u, _v;      // координаты текстуры

     static const DWORD FVF;
};
const DWORD Vertex::FVF = D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1;
```

Обратите внимание, что к описанию формата вершины мы добавили константу **D3DFVF_TEX1**, которая говорит о том, что наша структура данных вершины содержит пару координат текстуры.

Теперь для каждого треугольника, образованного тремя объектами **Vertex** также указывается соответствующий треугольный фрагмент текстуры, заданный с помощью координат текстуры.

ПРИМЕЧАНИЕ Хотя мы задаем соответствие фрагмента текстуры и треугольной грани в трехмерном пространстве, наложение текстур не выполняется до этапа растеризации, на котором треугольная грань в трехмерном пространстве уже преобразована в пространство экрана.

6.2 Создание текстур и разрешение текстурирования

Данные текстур обычно считываются из хранящихся на диске файлов изображений и загружаются в объект **IDirect3DTexture9**. Для этого используется следующая функция библиотеки D3DX:

```
HRESULT D3DXCreateTextureFromFile(
     LPDIRECT3DDEVICE9 pDevice,     // устройство для создания
                                    // текстуры
     LPCSTR pSrcFile,               // имя файла с загружаемым
                                    // изображением
     LPDIRECT3DTEXTURE9* ppTexture // указатель для возврата
                                    // созданной текстуры
);
```

Данная функция позволяет загружать изображения в форматах BMP, DDS, DIB, JPG, PNG и TGA.

Например, чтобы создать текстуру из изображения, хранящегося в файле с именем stonewall.bmp, следует использовать такой код:

```
IDirect3Dtexture9* _stonewall;
D3DXCreateTextureFromFile(_device, "stonewall.bmp", &_stonewall);
```

Для установки текущей текстуры используется следующий метод:

```
HRESULT IDirect3DDevice9::SetTexture(
     DWORD Stage,                    // Значение в диапазоне 0-7,
                                     // идентифицирующее этап
                                     // текстурирования
                                     // (см. примечание ниже)
     IDirect3DBaseTexture9* pTexture // Указатель на устанавливаемую
                                     // текстуру
);
```

Вот пример использования данного метода:

```
Device->SetTexture(0, _stonewall);
```

ПРИМЕЧАНИЕ В Direct3D вы можете устанавливать до восьми текстур, которые будут объединяться для получения более детализированного изображения. Этот процесс называется *мультитекстурированием* (*multitexturing*). В этой книге до четвертой части мы не будем пользоваться мультитекстурированием, и поэтому номер этапа текстурирования будет равен 0.

Чтобы запретить наложение текстуры на конкретном этапе текстурирования следует установить значение **`pTexture`** для соответствующего этапа равным 0.

Например, если мы не хотим использовать при отображении объекта текстуры, следует написать:

```
Device->SetTexture(0, 0);
renderObjectWithoutTexture();
```

Если в нашей сцене присутствуют треугольники, для которых используются различные текстуры, код должен выглядеть похожим на приведенный ниже фрагмент:

```
Device->SetTexture(0, _tex0);
drawTrisUsingTex0();

Device->SetTexture(0, _tex1);
drawTrisUsingTex1();
```

6.3 Фильтры

Как упоминалось ранее, текстуры накладываются на треугольники в пространстве экрана. Обычно размер треугольного фрагмента текстуры отличается от размера треугольной грани на экране. Когда фрагмент текстуры меньше, чем изображение грани на экране, выполняется увеличение фрагмента текстуры до размеров грани. Когда текстура больше, чем грань на экране, она сжимается. И в том и в другом случае возникают искажения. *Фильтрацией* (*filtering*) называется используемая в Direct3D техника уменьшения этих искажений.

Direct3D предоставляет три различных метода фильтрации, от выбора которых зависит качество итогового изображения. Чем выше качество, тем меньше скорость визуализации, так что вам придется выбирать между качеством и скоростью. Устанавливаются фильтры текстур с помощью метода **`IDirect3DDevice9::SetSamplerState`**.

- **Выборка ближайшей точки** (nearest point sampling) — Это используемый по умолчанию метод фильтрации, обеспечивающий грубое приближение, но требующий минимум вычислительных ресурсов. Чтобы использовать данный фильтр для увеличения и уменьшения текстур, добавьте в приложение следующий код:

```
Device->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_POINT);
Device->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_POINT);
```

- **Линейная фильтрация** (linear filtering) — Этот тип фильтра обеспечивает хорошее качество изображения и на современном оборудовании выполняется достаточно быстро. Рекомендуется использовать в качестве минимального варианта именно этот метод. Чтобы использовать для увеличения и уменьшения текстур линейную фильтрацию, добавьте в приложение следующий код:

```
Device->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
Device->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
```

- **Анизотропная фильтрация** (anisotropic filtering) — Этот тип фильтра обеспечивает наилучшее качество изображения, но при этом требует значительного объема вычислений. Чтобы использовать для увеличения и уменьшения текстур анизотропную фильтрацию, добавьте в приложение следующий код:

```
Device->SetSamplerState(0, D3DSAMP_MAGFILTER,
                           D3DTEXF_ANISOTROPIC);
Device->SetSamplerState(0, D3DSAMP_MINFILTER,
                           D3DTEXF_ANISOTROPIC);
```

Если вы используете анизотропную фильтрацию, необходимо также задать уровень фильтрации **`D3DSAMP_MAXANISOTROPY`**, определяющий качество фильтрации. Чем больше значение этого параметра, тем лучше результат. Чтобы узнать диапазон значений, поддерживаемый установленным оборудованием, необходимо проверить структуру **`D3DCAPS9`**. Приведенный ниже фрагмент кода устанавливает значение уровня фильтрации равным 4:

```
Device->SetSamplerState(0, D3DSAMP_MAXANISOTROPY, 4);
```

6.4 Детализируемые текстуры

Как говорилось в разделе 6.3, треугольник на экране зачастую не совпадает по размерам с треугольным фрагментом текстуры. Чтобы уменьшить различие

размеров, можно создать *цепочку текстур с постепенно понижаемой детализацией* (*mipmap*). Идея состоит в том, чтобы взять текстуру и на ее основе создать ряд изображений меньшего размера и с меньшим разрешением и для каждого из этих уровней индивидуально настроить фильтрацию, чтобы на изображении сохранялись важные для нас детали (рис. 6.4).

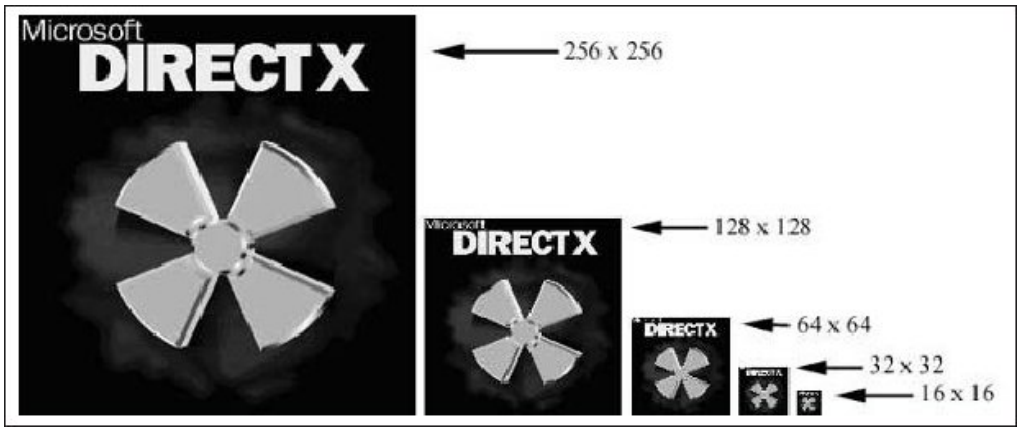

Рис. 6.4. *Цепочка текстур с понижаемой детализацией; обратите внимание, что каждое последующее изображение в цепочке в два раза меньше предыдущего*

6.4.1 Фильтр детализации текстур

Фильтр детализации текстур используется для того чтобы управлять тем, как Direct3D использует детализируемые текстуры. Чтобы установить этот фильтр можно написать:

```
Device->SetSamplerState(0, D3DSAMP_MIPFILTER, Filter);
```

где **`Filter`** может принимать одно из следующих значений:

- **`D3DTEXF_NONE`** — Детализация выключена.
- **`D3DTEXF_POINT`** — При использовании этого фильтра Direct3D выбирает тот уровень детализации, который наиболее точно соответсвует размеру треугольника на экране. После выбора наиболее подходящей текстуры, Direct3D применяет к ней установленные фильтры для увеличения или уменьшения.
- **`D3DTEXF_LINEAR`** — При использовании этого фильтра Direct3D выбирает два уровня детализации, которые наиболее точно соответсвуют размеру треугольника на экране, применяет к ним установленные фильтры для увеличения или уменьшения и выполняет линейную интерполяцию двух уровней для получения итогового значения цвета.

6.4.2 Использование детализируемых текстур в Direct3D

Использовать детализируемые текстуры в Direct3D очень просто. Если видеокарта поддерживает детализацию текстур, то функция **`D3DXCreateTextureFromFile`** сгенерирует цепочку текстур с понижаемой детализацией за вас. Кроме того, Direct3D автоматически выбирает то изображение из цепочки, которое наилучшим образом соответствует треугольнику на экране. Поэтому детализация текстур используется почти всегда и устанавливается автоматически.

6.5 Режимы адресации

Раньше мы утверждали, что координаты текстур должны находиться в диапазоне [0, 1]. С технической точки зрения это неверно, и координаты могут выходить за указанный диапазон. Поведение Direct3D в том случае, если координаты выходят за диапазон [0, 1] определяется установленным режимом адресации. Поддерживаются четыре режима адресаци: *обертывание* (*wrap*), *цвет рамки* (*border color*), *одиночное наложение* (*clamp*) и *отражение* (*mirror*), которые показаны на рис. 6.5, 6.6, 6.7 и 6.8 соответственно.

Рис. 6.5. *Режим обертывания*

Рис. 6.6. *Режим цвета рамки*

Рис. 6.7. *Режим одиночного наложения*

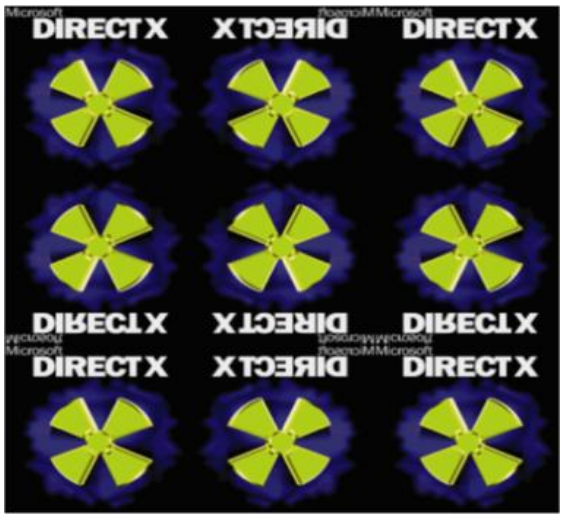

Рис. 6.8. *Режим отражения*

На этих рисунках координаты текстур для четырех вершин квадрата определены следующим образом: (0, 0), (0, 3), (3, 0) и (3, 3). Поскольку по осям U и V размер равен трем единицам, квадрат делится на матрицу 3 × 3. Если, к примеру, вы хотите, чтобы текстура накладывалась в виде матрицы размером 5 × 5, включите режим обертывания и задайте координаты текстур (0, 0), (0, 5), (5, 0) и (5, 5).

Приведенный ниже фрагмент кода взят из программы AddressModes и показывает, как устанавливаются четыре режима адресации:

```
// Установка режима обертывания
if (::GetAsyncKeyState('W') & 0x8000f)
{
    Device->SetSamplerState(0, D3DSAMP_ADDRESSU, D3DTADDRESS_WRAP);
    Device->SetSamplerState(0, D3DSAMP_ADDRESSV, D3DTADDRESS_WRAP);
}
```

```
// Установка режима цвета рамки
if (::GetAsyncKeyState('B') & 0x8000f)
{
    Device->SetSamplerState(0, D3DSAMP_ADDRESSU, D3DTADDRESS_BORDER);
    Device->SetSamplerState(0, D3DSAMP_ADDRESSV, D3DTADDRESS_BORDER);
    Device->SetSamplerState(0, D3DSAMP_BORDERCOLOR, 0x000000ff);
}

// Установка режима отсечения
if (::GetAsyncKeyState('C') & 0x8000f)
{
    Device->SetSamplerState(0, D3DSAMP_ADDRESSU, D3DTADDRESS_CLAMP);
    Device->SetSamplerState(0, D3DSAMP_ADDRESSV, D3DTADDRESS_CLAMP);
}

// Установка режима отражения
if (::GetAsyncKeyState('M') & 0x8000f)
{
    Device->SetSamplerState(0, D3DSAMP_ADDRESSU, D3DTADDRESS_MIRROR);
    Device->SetSamplerState(0, D3DSAMP_ADDRESSV, D3DTADDRESS_MIRROR);
}
```

6.6 Пример приложения: текстурированный квадрат

Пример приложения из этой главы показывает как текстурировать квадрат и установить фильтр текстуры (рис. 6.9). Если видеокарта поддерживает детализацию текстур, при загрузке текстуры функцией **`D3DXCreateTextureFromFile`** будет автоматически создана цепочка текстур с понижающейся детализацией.

Рис. 6.9. *Окно с изображением текстурированого квадрата, полученное в приложении TexQuad*

ПРИМЕЧАНИЕ В сопроводительных файлах есть еще два примера приложений для данной главы. Один пример отображает куб с наложенной текстурой деревянного ящика (рис. 6.1). Другой пример демонстрирует различные режимы адресации.

Для добавления к сцене текстур объектов необходимо выполнить следующие действия:

1. Создать вершины объектов, содержащие заданные координаты текстур.
2. Загрузить текстуру в интерфейс **IDirect3DTexture9** с помошью метода **D3DXCreateTextureFromFile**.
3. Установить фильтры для увеличения, уменьшения и детализации текстур.
4. Перед тем, как рисовать объект, указать связанную с объектом текстуру с помощью метода **IDirect3DDevice9::SetTexture**.

Мы начинаем с объявления нескольких глобальных переменных — одной для глобального буфера вершин, хранящего данные вершин квадрата, и другой для текстуры, которая будет накладываться на квадрат:

```
IDirect3DVertexBuffer9* Quad = 0;
IDirect3DTexture9*       Tex = 0;
```

Функция **Setup** достаточно прямолинейна; мы создаем квадрат из двух треугольников и задаем для них координаты текстуры. Затем мы загружаем файл с растровым изображением dx5_logo.bmp в интерфейс **IDirect3DTexture9**. Теперь мы можем разрешить использование текстур с помощью метода **SetTexture**. После вышеописанных действий мы указываем, что для уменьшения и увеличения текстур используется линейная фильтрация, и устанавливаем фильтр детализации текстур **D3DTEXF_POINT**:

```
bool Setup()
{
   Device->CreateVertexBuffer(
                    6 * sizeof(Vertex),
                    D3DUSAGE_WRITEONLY,
                    Vertex::FVF,
                    D3DPOOL_MANAGED,
                    &Quad,
                    0);

   Vertex* v;
   Quad->Lock(0, 0, (void**)&v, 0);

   // Квадрат состоит из двух треугольников,
   // обратите внимание на координаты текстур:
   v[0] = Vertex(-1.0f, -1.0f, 1.25f, 0.0f, 0.0f, -1.0f, 0.0f, 1.0f);
   v[1] = Vertex(-1.0f,  1.0f, 1.25f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f);
   v[2] = Vertex( 1.0f,  1.0f, 1.25f, 0.0f, 0.0f, -1.0f, 1.0f, 0.0f);

   v[3] = Vertex(-1.0f, -1.0f, 1.25f, 0.0f, 0.0f, -1.0f, 0.0f, 1.0f);
   v[4] = Vertex( 1.0f,  1.0f, 1.25f, 0.0f, 0.0f, -1.0f, 1.0f, 0.0f);
   v[5] = Vertex( 1.0f, -1.0f, 1.25f, 0.0f, 0.0f, -1.0f, 1.0f, 1.0f);
```

```
    Quad->Unlock();

    // Загрузка данных текстуры
    D3DXCreateTextureFromFile(
                    Device,
                    "dx5_logo.bmp",
                    &Tex);

    // Разрешаем текстурирование
    Device->SetTexture(0, Tex);

    // Устанавливаем фильтры текстуры
    Device->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
    Device->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
    Device->SetSamplerState(0, D3DSAMP_MIPFILTER, D3DTEXF_POINT);

    // Задаем матрицу проекции
    D3DXMATRIX proj;
    D3DXMatrixPerspectiveFovLH(
                    &proj,
                    D3DX_PI * 0.5f, // 90 градусов
                    (float)Width / (float)Height,
                    1.0f,
                    1000.0f);
    Device->SetTransform(D3DTS_PROJECTION, &proj);

    // Не использовать в этом примере освещение
    Device->SetRenderState(D3DRS_LIGHTING, false);

    return true;
}
```

Теперь мы можем визуализировать наш квадрат обычным способом и на его будет наложена указанная текстура:

```
bool Display(float timeDelta)
{
     if (Device)
     {
          Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                        0xffffffff, 1.0f, 0);
          Device->BeginScene();

          Device->SetStreamSource(0, Quad, 0, sizeof(Vertex));
          Device->SetFVF(Vertex::FVF);

          // Рисуем примитив с наложением указанной текстуры
          Device->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 2);

          Device->EndScene();
          Device->Present(0, 0, 0, 0);
     }
     return true;
}
```

6.7 Итоги

- Координаты текстуры используются для задания треугольного фрагмента текстуры, который будет наложен на треугольную грань трехмерного объекта.
- Мы можем создавать текстуры из хранящихся на диске файлов изображений с помощью метода **`D3DXCreateTextureFromFile`**.
- Фильтрация текстур выполняется путем задания фильтров для увеличения, уменьшения и детализации текстуры.
- Режим адресации определяет поведение Direct3D в тех случаях, когда координаты текстуры выходят за диапазон [0, 1]. Текстура может копироваться, отражаться, обрезаться и т.д.

Глава 7
Смешивание

В этой главе мы изучим технику, называемую *смешивание* (*blending*), которая позволяет смешивать (комбинировать) цвет растеризуемого в данный момент пикселя с цветами ранее растеризованных в том же самом месте пикселей. Другими словами, мы смешиваем текущий примитив с ранее нарисованными примитивами. Эта техника позволяет реализовать ряд эффектов (в частности, прозрачность).

Цели

- Понять, как работает смешивание и как его можно использовать.
- Изучить различные режимы смешивания, поддерживаемые Direct3D.
- Посмотреть как альфа-составляющая используется для управления прозрачностью примитивов.

7.1 Формулы смешивания

Взгляните на рис. 7.1, где красный чайник изображен поверх фоновой картинки с текстурой деревянного ящика.

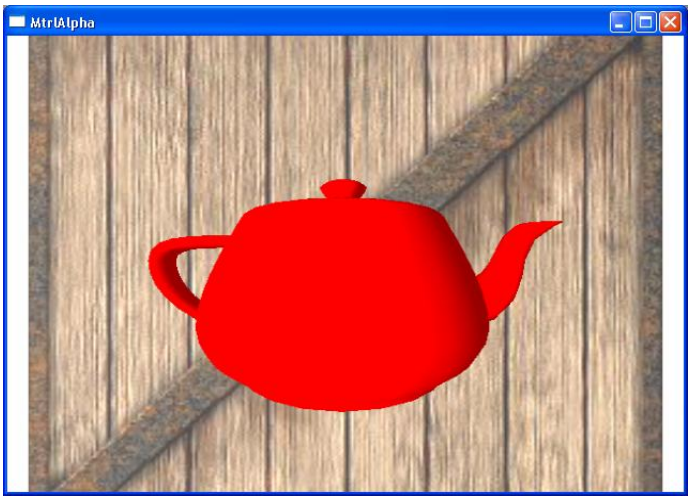

Рис. 7.1. *Непрозрачный чайник*

Предположим, мы хотим нарисовать чайник с заданным уровнем прозрачности, чтобы сквозь него была видна фоновая текстура с изображением ящика (рис. 7.2).

Рис. 7.2. *Прозрачный чайник*

Как выполнить это? Поскольку мы растеризуем образующие чайник треугольники поверх изображения ящика, мы должны комбинировать цвета пикселей чайника с цветами пикселей ящика таким образом, чтобы ящик был виден сквозь чайник.

Комбинирование значений рисуемых в данный момент пикселей (пиксели источника) со значениями ранее записанных пикселей (пиксели приемника) и называется *смешиванием*. Обратите внимание, что реализуемые с помощью смешивания эффекты не ограничиваются обычной иммитацией прозрачности стеклянных объектов. Мы можем указать ряд параметров, определяющих как будет выполняться смешивание. Эти параметры рассматриваются в разделе 7.2.

Важно понимать, что пиксели растеризуемых в данный момент треугольников смешиваются с пикселями, которые до этого были помещены во вторичный буфер. В рассматриваемом примере сперва был нарисован ящик, и его пиксели были помещены во вторичный буфер. Затем мы нарисовали чайник и его пиксели смешивались с пикселями ящика. Следовательно, при использовании смешивания необходимо руководствоваться следующим правилом:

ПРАВИЛО *Сперва рисуйте объекты, которые не используют смешивание. Затем отсортируйте объекты, которые используют смешивание, по расстоянию от камеры; наиболее эффективно эта операция выполняется, если объекты находятся в пространстве вида — в этом случае достаточно отсортировать их по значению координаты Z. После этого рисуйте использующие смешивание объекты начиная от самых дальних и заканчивая самыми близкими.*

При смешивании значений двух пикселей используется следующая формула:

ИтоговыйПиксель =
ПиксельИсточника ⊗ *КоэффициентСмешиванияИсточника* +
ПиксельПриемника ⊗ *КоэффициентСмешиванияПриемника*

Каждая из переменных в этой формуле является четырехмерным цветовым вектором (r, g, b, a), а символ ⊗ означает операцию перемножения компонент.

- *ИтоговыйПиксель* — Пиксель, получаемый в результате смешивания.
- *ПиксельИсточника* — Обрабатываемый в данный момент пиксель, который смешивается с пикселем из вторичного буфера.
- *КоэффициентСмешиванияИсточника* — Значение в диапазоне [0, 1], определяющее какой процент пикселя источника участвует в смешивании.
- *Пиксель приемника* — Пиксель, находящийся во вторичном буфере.
- *КоэффициентСмешиванияПриемника* — Значение в диапазоне [0, 1], определяющее какой процент пикселя приемника участвует в смешивании.

Благодаря коэффициентам смешивания источника и приемника можно различными способами модифицировать исходные пиксели источника и приемника, что позволяет реализовать различные эффекты. В разделе 7.2 описаны предопределенные значения, которые можно использовать.

По умолчанию смешивание запрещено; чтобы разрешить его, присвойте режиму визуализации **`D3DRS_ALPHABLENDENABLE`** значение **`true`**:

```
Device->SetRenderState(D3DRS_ALPHABLENDENABLE, true);
```

СОВЕТ Смешивание — достаточно сложная и ресурсоемкая операция и ее надо разрешать только для тех объектов, которым она необходима. Закончив визуализацию этих объектов, смешивание следует выключить. Также попытайтесь объединить все треугольники, для которых необходимо смешивание и визуализировать их за один раз, чтобы избежать многократных включений и выключений смешивания в одном кадре.

7.2 Коэффициеты смешивания

Задавая различные комбинации коэффициентов смешивания источника и приемника вы можете реализовать десятки различных эффектов. Поэкспериментируйте с различными комбинациями, чтобы увидеть что они делают. Чтобы установить коэффициент смешивания источника и коэффициент смешивания приемника надо задать значения режимов визуализации **D3DRS_SRCBLEND** и **D3DRS_DESTBLEND** соответственно. Например, мы можем написать:

```
Device->SetRenderState(D3DRS_SRCBLEND, Source);
Device->SetRenderState(D3DRS_DESTBLEND, Destination);
```

где **Source** и **Destination** могут принимать значения одного из следующих коэффициентов смешивания:

- **D3DBLEND_ZERO** — коэффициент смешивания = (0, 0, 0, 0)
- **D3DBLEND_ONE** — коэффициент смешивания = (1, 1, 1, 1)
- **D3DBLEND_SRCCOLOR** — коэффициент смешивания = (r_s, g_s, b_s, a_s)
- **D3DBLEND_INVSRCCOLOR** — коэффициент смешивания = ($1 - r_s$, $1 - g_s$, $1 - b_s$, $1 - a_s$)
- **D3DBLEND_SRCALPHA** — коэффициент смешивания = (a_s, a_s, a_s, a_s)
- **D3DBLEND_INVSRCALPHA** — коэффициент смешивания = ($1 - a_s$, $1 - a_s$, $1 - a_s$, $1 - a_s$)
- **D3DBLEND_DESTALPHA** — коэффициент смешивания = (a_d, a_d, a_d, a_d)
- **D3DBLEND_INVDESTALPHA** — коэффициент смешивания = ($1 - a_d$, $1 - a_d$, $1 - a_d$, $1 - a_d$)
- **D3DBLEND_DESTCOLOR** — коэффициент смешивания = (r_d, g_d, b_d, a_d)
- **D3DBLEND_INVDESTCOLOR** — коэффициент смешивания = ($1 - r_d$, $1 - g_d$, $1 - b_d$, $1 - a_d$)
- **D3DBLEND_SRCALPHASAT** — коэффициент смешивания = (f,f,f, 1), где $f = \min(a_s, 1 - a_d)$
- **D3DBLEND_BOTHINVSRCALPHA** — Этот режим смешивания устанавливает коэффициент смешивания источника равным ($1 - a_s$, $1 - a_s$, $1 - a_s$, $1 - a_s$), а коэффициент смешивания приемника равным

(a_s, a_s, a_s, a_s). Его можно указывать только для режима визуализации **D3DRS_SRCBLEND**.

Значениями по умолчанию для коэффициента смешивания источника и коэффициента смешивания приемника являются **D3DBLEND_SRCALPHA** и **D3DBLEND_INVSRCALPHA** соответственно.

7.3 Прозрачность

В предыдущей главе мы игнорировали альфа-компоненту цвета вершины и материала потому что они были нам не нужны так как используются в основном для смешивания. Тем не менее, при затенении треугольной грани альфа-компоненты каждой из вершин комбинируются для получения альфа-компоненты каждого пикселя точно так же, как для получения цвета пикселя комбинируются цвета вершин.

Альфа-компонента в основном используется для задания степени прозрачности пикселя. Предположим, что для альфа-компоненты каждого пикселя выделено 8 бит. Тогда диапазон значений альфа-компоненты будет [0, 255], где [0, 255] соответствует [0%, 100%] непрозрачности. Черный пиксель альфа-канала (со значением 0) полностью прозрачен, серый пиксель альфа-канала (со значением 128) прозрачен на 50%, а белый пиксель альфа-канала (со значением 255) полностью непрозрачен.

Чтобы альфа-компонента задавала уровень прозрачности пикселей, мы должны присвоить коэффициенту смешивания источника значение **D3DBLEND_SRCALPHA** а коэффициенту смешивания приемника значение **D3DBLEND_INVSRCALPHA**. Эти значения являются устанавливаемыми по умолчанию коэффициентами смешивания.

7.3.1 Альфа-канал

Вместо того, чтобы вычислять альфа-компоненту при затенении, можно получать ее значение из *альфа-канала* (*alpha channel*) текстуры. Альфа-каналом называется дополнительный набор битов, резервируемых для каждого текселя, в которых кранится альфа-компонент. Когда текстура накладывается на примитив также накладываются и альфа-компоненты из альфа-канала и они становятся значениями альфа-компоненты пикселей текстурированного примитива. На рис. 7.3 представлено изображение 8-разрядного альфа-канала.

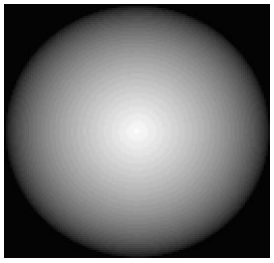

Рис. 7.3. *8-разрядная карта оттенков серого, представляющая альфа-канал текстуры*

На рис. 7.4 показан результат визуализации текстурированного квадрата с альфа-каналом, определяющим какие части будут прозрачными.

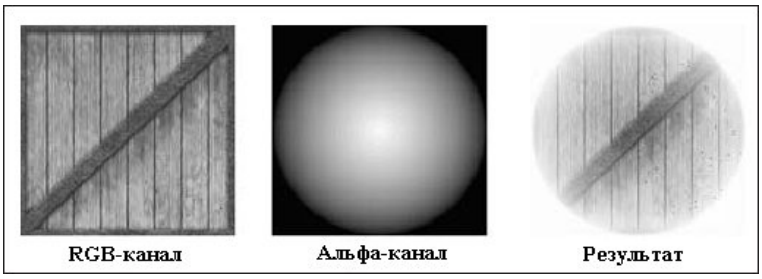

Рис. 7.4. *Текстурированный квадрат, у которого альфа-канал задает прозрачность отдельных частей*

7.3.2 Указание источника альфа-компоненты

Если у используемой в данный момент текстуры есть альфа-канал, то по умолчанию значения альфа-компоненты пикселей берутся из него. Если альфа-канала нет, значения альфа-компонент отдельных пикселей вычисляются на основании значений альфа-компонент вершин. В то же время с помощью показанных ниже режимов визуализации вы сами можете указать, какой источник альфа компоненты использовать (цвета вершин или альфа-канал):

```
// Вычисляем альфа-компоненту при затенении
// на основании цветов вершин
Device->SetTextureStageState(0, D3DTSS_ALPHAARG1, D3DTA_DIFFUSE);
Device->SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1);

// Берем значение альфа-компоненты из альфа-канала
Device->SetTextureStageState(0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE);
Device->SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1);
```

7.4 Создание альфа-канала с помощью утилиты DirectX Texture Tool

Большинство распространенных форматов графических файлов не хранят информацию альфа-компонент. В этом разделе мы покажем вам как создать файл формата DDS с альфа-каналом используя утилиту DirectX Texture Tool. DDS — это формат графических файлов специально разработанный для приложений и текстур DirectX. Файлы DDS могут быть загружены в объект текстуры с помощью функции **`D3DXCreateTextureFromFile`** точно так же как файлы BMP и JPG. Утилита DirectX Texture Tool находится в папке \Bin\DXUtils корневого каталога DXSDK.

Запустите утилиту DirectX Texture Tool и откройте файл crate.jpg, находящийся на сопроводительном компакт-диске в папке с примерами программ к данной главе. Изображение ящика автоматически загружается как 24-разрядная RGB-текстура, в которой у каждого пикселя 8 бит отведено под красный цвет, 8 бит — под зеленый и 8 бит — под синий. Нам надо преобразовать эту текстуру в 32-разрядную ARGB-текстуру, зарезервировав дополнительные 8 бит для альфа-канала. Выберите в меню **Format** команду **Change Surface Format**. Будет выведено диалоговое окно, показанное на рис. 7.5. Выберите формат **A8 R8 G8 B8** и щелкните по кнопке **OK**.

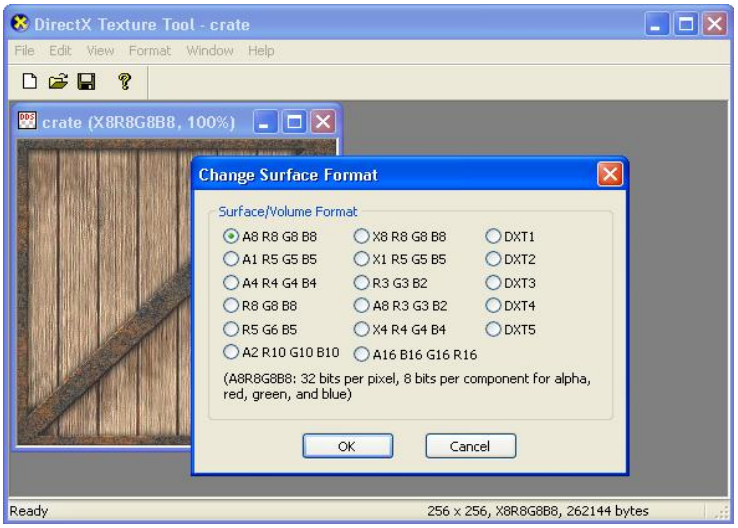

Рис. 7.5. *Изменение формата текстуры*

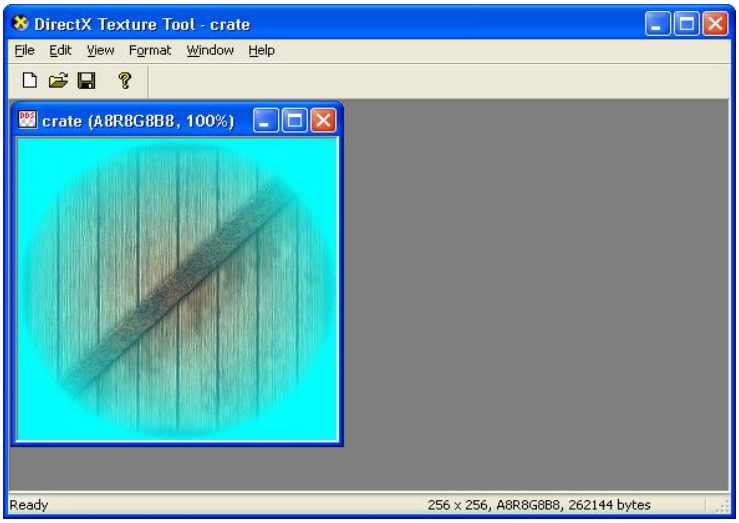

Рис. 7.6. *Полученная текстура с альфа-каналом*

В результате будет создано изображение с 32-разрядной глубиной цвета, где у каждого пикселя 8 бит отведено под альфа-канал, 8 бит для красного цвета, 8 бит для зеленого и 8 бит для синего. Следующая задача — загрузить данные в альфа-канал. Мы будем загружать в альфа-канал 8-разрядную карту оттенков серого, изображенную на рис. 7.3. Раскройте меню **File** и выберите пункт **Open Onto Alpha Channel Of This Texture**. На экран будет выведено диалоговое окно, предлагающее выбрать файл изображения, содержащий данные, которые вы хотите загрузить в альфа-канал. Выберите файл alphachannel.bmp расположенный в папке примера к этой главе texAlpha. На рис. 7.6 показано окно программы после загрузки данных альфа-канала.

Теперь можно сохранить текстуру в файле; мы выбрали для файла имя cratewalpha.dds.

7.5 Пример приложения: прозрачный чайник

Пример приложения, который мы будем рассматривать, рисует прозрачный чайник поверх фоновой текстуры с изображением ящика, как показано на рис. 7.2. Данные альфа-компоненты в этом примере берутся из материала. Приложение позволяет увеличивать и уменьшать значение альфа-компоненты нажатием на клавиши **A** и **S**. Нажатие на клавишу **A** увеличивает значение альфа-компоненты; нажатие на клавишу **S** — уменьшает его.

Чтобы использовать смешивание необходимо выполнить следующие действия:

1. Установить коэффициенты смешивания **D3DRS_SRCBLEND** и **D3DRS_DESTBLEND**.
2. Если используется альфа-компонента, указать ее источник (материал или альфа-канал текстуры).
3. Установить режим визуализации с альфа-смешиванием.

В примере мы объявляем несколько самодокументируемых глобальных переменных:

```
ID3DXMesh* Teapot = 0;   // чайник
D3DMATERIAL9 TeapotMtrl; // материал чайника

IDirect3DVertexBuffer9* BkGndQuad = 0; // квадрат фона
IDirect3DTexture9* BkGndTex = 0;       // текстура ящика
D3DMATERIAL9 BkGndMtrl;                // материал фона
```

Метод **Setup** делает много вещей, но мы опустим большую часть кода, которая не относится к рассматриваемой в этой главе теме. Что касается смешивания, метод **Setup** задает источник из которого будут браться значения альфа-компонент. В рассматриваемом примере мы указываем, что значения альфа-компонент будут браться из соответствующей компоненты материала. Обратите

внимание, что для материала чайника мы задаем значение альфа-компоненты равное 0.5, а это значит, что чайник будет визуализирован с 50% прозрачностью. Помимо вышеперечисленных действий мы также задаем коэффициенты смешивания. Обратите внимание, что в этом методе мы не разрешаем альфа-смешивание. Дело в том, что альфа-смешивание — это ресурсоемкая операция, которая должна включаться только при визуализации тех объектов, для которых она нужна. Например, в рассматриваемой программе визуализация с альфа-смешиванием нужна только для чайника и не требуется для квадрата с фоновой текстурой. Поэтому мы разрешаем альфа-смешивание в функции **Display**.

```
bool Setup()
{
     TeapotMtrl = d3d::RED_MTRL;
     TeapotMtrl.Diffuse.a = 0.5f; // 50% прозрачность
     BkGndMtrl = d3d::WHITE_MTRL;

     D3DXCreateTeapot(Device, &Teapot, 0);

... // Код создания квадрата фона пропущен

... // Код установки освещения и текстур пропущен

     // В качестве источника альфа-компоненты используем
     // параметры материала
     Device->SetTextureStageState(0, D3DTSS_ALPHAARG1,
                                  D3DTA_DIFFUSE);
     Device->SetTextureStageState(0, D3DTSS_ALPHAOP,
                                  D3DTOP_SELECTARG1);

     // Устанавливаем коэффициенты смешивания таким образом,
     // чтобы альфа-компонента определяла прозрачность
     Device->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
     Device->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);

... // установка матриц проекции/вида пропущена

     return true;
}
```

В функции **Display** мы проверяем нажаты ли клавиши **A** или **S** и, если да, то соответственно увеличиваем или уменьшаем значение альфа-компоненты материала. Обратите внимание, что метод гарантирует, что значение альфа-компоненты не выйдет за пределы диапазона [0, 1]. Затем мы разрешаем альфа-смешивание, визуализируем чайник с включенным альфа-смешиванием, после чего выключаем альфа-смешивание.

```
bool Display(float timeDelta)
{
     if(Device)
     {
```

```
        //
        // Обновление
        //

        // Увеличение/уменьшение альфа-компоненты
        // с помощью клавиатуры
        if(::GetAsyncKeyState('A') & 0x8000f )
             TeapotMtrl.Diffuse.a += 0.01f;
        if( ::GetAsyncKeyState('S') & 0x8000f )
             TeapotMtrl.Diffuse.a -= 0.01f;

        // Проверяем не вышло ли значение за интервал [0, 1]
        if(TeapotMtrl.Diffuse.a > 1.0f)
             TeapotMtrl.Diffuse.a = 1.0f;
        if(TeapotMtrl.Diffuse.a < 0.0f)
             TeapotMtrl.Diffuse.a = 0.0f;


        //
        // Визуализация
        //

        Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                     0xffffffff, 1.0f, 0);
        Device->BeginScene();

        // Рисуем фон
        D3DXMATRIX W;

        D3DXMatrixIdentity(&W);
        Device->SetTransform(D3DTS_WORLD, &W);
        Device->SetFVF(Vertex::FVF);
        Device->SetStreamSource(0, BkGndQuad, 0, sizeof(Vertex));
        Device->SetMaterial(&BkGndMtrl);
        Device->SetTexture(0, BkGndTex);
        Device->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 2);

        // Рисуем чайник
        Device->SetRenderState(D3DRS_ALPHABLENDENABLE, true);

        D3DXMatrixScaling(&W, 1.5f, 1.5f, 1.5f);
        Device->SetTransform(D3DTS_WORLD, &W);
        Device->SetMaterial(&TeapotMtrl);
        Device->SetTexture(0, 0);
        Teapot->DrawSubset(0);

        Device->SetRenderState(D3DRS_ALPHABLENDENABLE, false);

        Device->EndScene();
        Device->Present(0, 0, 0, 0);
    }
    return true;
}
```

ПРИМЕЧАНИЕ На веб-сайте этой книги есть еще один пример к данной главе, texAlpha, который демонстрирует использование альфа-канала текстуры. Единственное отлтчие кода этого примера от рассмотренного выше заключается в том, что мы указываем в качестве источника альфа-компоненты не материал, а альфа-канал текстуры.

```
// Использовать альфа-канал в качестве
// источника альфа-компонент
Device->SetTextureStageState(0, D3DTSS_ALPHAARG1,
                             D3DTA_TEXTURE);
Device->SetTextureStageState(0, D3DTSS_ALPHAOP,
                             D3DTOP_SELECTARG1);
```

Приложение загружает файл DDS, содержащий альфа-канал, созданный с помощью утилиты DX Tex Tool, рассмотренной в разделе 7.4.

7.6 Итоги

- Альфа-смешивание позволяет комбинировать пиксель растеризуемого в данный момент примитива с пикселом уже находящимся в той же самой позиции вторичного буфера.
- Коэффициенты смешивания позволяют нам управлять тем, как будут объединяться пиксели источника и приемника.
- Значения альфа-компоненты могут браться из альфа-компоненты материала примитива, либо из альфа-канала накладываемой на примитив текстуры.

Глава 8
Трафареты

Эта глава знакомит нас с *буфером трафарета* (*stencil buffer*) и завершает вторую часть книги. Буфер трафарета является внеэкранным буфером, который можно использовать для реализации ряда спецэффектов. Буфер трафарета имеет то же самое разрешение, что вторичный буфер и буфер глубины, так что пиксел буфера трафарета, находящийся в позиции (i, j) соответствует пикселю в позиции (i, j) во вторичном буфере и буфере глубины. Как видно из названия, буфер трафарета действует подобно трафарету, и позволяет блокировать визуализацию отдельных частей вторичного буфера.

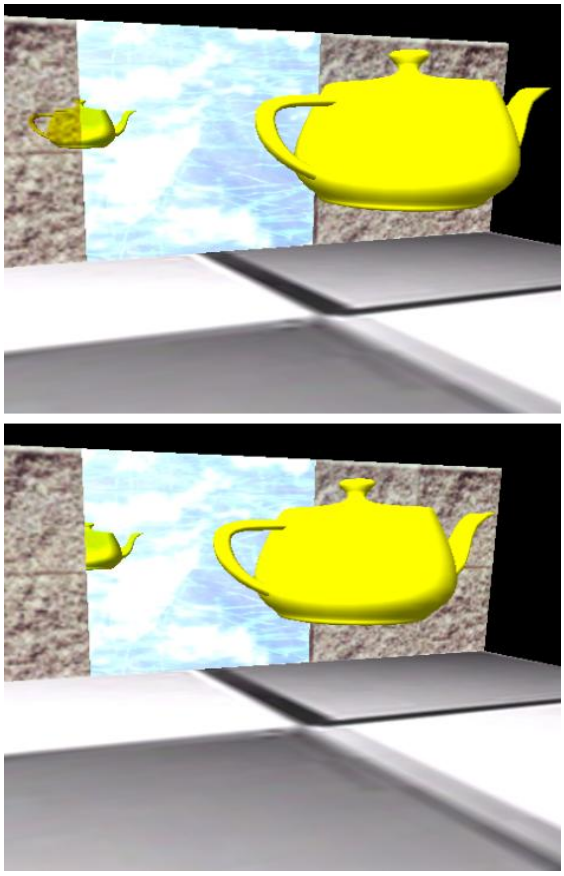

Рис. 8.1. *На рисунке сверху изображено отражение чайника без использования буфера трафарета. Видно, что отражение визуализируется независимо от того, где оно находится — в зеркале или на стене. Используя буфер трафарета мы можем блокировать визуализацию тех частей отражения чайника, которые не попадают в зеркало (рисунок снизу)*

Например, для того чтобы реализовать зеркало, нам просто необходимо отразить объект относительно плоскости зеркала; однако, вы наверняка хотите, чтобы отражение изображалось только в зеркале. Мы можем использовать буфер трафарета, чтобы блокировать визуализацию тех частей отражения, которые расположены вне зеркала. Эта ситуация поясняется на рис. 8.1.

Буфер трафарета является небольшой частью Direct3D и управляется через простой интерфейс. Подобно смешиванию, простой интерфейс предоставляет гибкий и мощный набор возможностей. Изучать использование буфера трафарета лучше всего на примере конкретных приложений. Рассмотрев применение буфера трафарета в нескольких приложениях, вы сможете лучше представлять область его применения в ваших собственных проектах. Поэтому в данной главе особый упор делается на изучение кода двух использующих трафареты программ (в частности, реализацию отражений и плоских теней).

Цели

- Получить представление о том, как работает буфер трафарета, как создать буфер трафарета и как управлять им.
- Узнать как создаются зеркальные отражения и как буфер трафарета используется для того, чтобы на поверхностях, не являющихся зеркальными, отражений не появлялось.
- Посмотреть, как с помощью буфера трафарета можно визуализировать тени и избежать «двойного смешивания».

8.1 Использование буфера трафарета

Чтобы работать с буфером трафарета мы должны сперва создать его при инициализации Direct3D, а затем разрешить его использование при визуализации. Создание буфера трафарета рассматривается в разделе 8.1.1. Чтобы разрешить использование буфера трафарета необходимо присвоить режиму визуализации **D3DRS_STENCILENABLE** значение **true**. Чтобы запретить использование буфера трафарета, присвойте режиму визуализации **D3DRS_STENCILENABLE** значение **false**. В приведенном ниже фрагменте кода мы сначала разрешаем использование буфера трафарета, а затем запрещаем его:

```
Device->SetRenderState(D3DRS_STENCILENABLE, true);

... // работа с буфером трафарета

Device->SetRenderState(D3DRS_STENCILENABLE, false);
```

ПРИМЕЧАНИЕ DirectX 9.0 поддерживает *двухсторонние трафареты* (*two-sided stencil*, в этой книге они не используются), которые ускоряют визуализацию теневых объемов благодаря уменьшению количества проходов визуализации, необходимых для рисования теневого объема. Подробнее об этом говорится в документации к SDK.

Мы можем очистить буфер трафарета, заполнив его указанным значением, с помощью метода **IDirect3DDevice9::Clear**. Вспомните, что тот же самый метод используется для очистки вторичного буфера и буфера глубины.

```
Device->Clear(0, 0,
     D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER | D3DCLEAR_STENCIL,
     0xff000000, 1.0f, 0 );
```

Обратите внимание, что в третьем аргументе мы добавили флаг **D3DCLEAR_STENCIL**, указывающий что мы хотим очистить буфер трафарета вместе со вторичным буфером и буфером глубины. Для задания значения, которым будет заполнен буфер трафарета используется шестой аргумент; в данном примере мы заполняем буфер нулями.

ПРИМЕЧАНИЕ Согласно презентации NVIDIA «Creating Reflections and Shadows Using Stencil Buffers» проведенной Марком Дж. Килгардом, в современных видеокартах, если вы используете буфер глубины, использование буфера трафарета можно рассматривать как «бесплатное приложение».

8.1.1 Создание буфера трафарета

Буфер трафарета создается в тот же самый момент, когда вы создаете буфер глубины. Задавая формат буфера глубины мы одновременно указываем и формат буфера трафарета. Фактически буфер глубины и буфер трафарета совместно используют один и тот же внеэкранный буфер поверхности, но каждый буфер испльзует свою часть выделяемой для каждого пикселя памяти. Для примера рассмотрим следующие три формата глубины/трафарета:

- **D3DFMT_D24S8** — Этот формат говорит, что создается 32-разрядный буфер глубины/трафарета в котором для каждого пикселя выделяются 24 бита под буфер глубины и 8 бит под буфер трафарета.
- **D3DFMT_D24X4S4** — Этот формат говорит, что создается 32-разрядный буфер глубины/трафарета в котором для каждого пикселя выделяются 24 бита под буфер глубины и 4 бита под буфер трафарета. Оставшиеся 4 разряда не используются.
- **D3DFMT_D15S1** — Этот формат говорит, что создается 16-разрядный буфер глубины/трафарета в котором для каждого пикселя выделяются 15 бит под буфер глубины и 1 бит под буфер трафарета.

Обратите внимание, что существуют форматы в которых разряды под буфер трафарета не выделяются вообще. Например, формат **D3DFMT_D32** говорит, что создается только 32-разрядный буфер глубины.

Кроме того, поддерживаемые форматы буфера трафарета различаются в зависимости от модели видеокарты. Например, некоторые видеокарты не поддерживают 8-разрядный буфер трафарета.

8.1.2 Проверка трафарета

Как упоминалось ранее, буфер трафарета можно использовать для блокирования визуализации отдельных частей вторичного буфера. Защищать конкретный пиксель от перезаписи или нет определяется с помощью проверки трафарета, выполняемой по следующей формуле:

```
(ref & mask) ОперацияСравнения (value & mask)
```

Если использование трафарета разрешено, проверка трафарета выполняется для каждого пикселя и в ней участвуют два операнда:

- Левый операнд (*LHS* = *ref* & *mask*) определяется путем выполнения поразрядной логической операции И между определенным в приложении эталонным значением (**ref**) и определенным в приложении значением маски (**mask**).
- Правый операнд (*RHS* = *value* & *mask*) определяется путем выполнения поразрядной логической операции И между соответствующим данному пикселю значением из буфера трафарета (**value**) и определенным в приложении значении маски (**mask**).

Затем в проверке трафарета сравниваются значения LHS и RHS; при этом используется заданная *ОперацияСравнения*. Результатом вычислений является

логическое значение (**true** или **false**). Мы записываем пиксель во вторичный буфер, если результатом проверки будет **true** (тест пройден). Если в результате проверки получается **false** (тест не пройден), пиксель не будет записан во вторичный буфер. Конечно, если пиксель не записывается во вторичный буфер, соответствующее ему значение в буфере глубины тоже не меняется.

8.1.3 Управление проверкой трафарета

Для увеличения гибкости Direct3D позволяет нам управлять переменными, используемыми в проверке трафарета. Другими словами, мы можем задавать эталонное значение трафарета, значение маски и даже операцию сравнения. Хотя мы не можем явно задать значение, записываемое в буфер трафарета, у нас есть достаточно возможностей для управления тем, какие значения будут записываться в буфер трафарета (и, кроме того, мы можем очистить буфер).

8.1.3.1 Эталонное значение трафарета

Эталонное значение трафарета **ref** по умолчанию равно нулю, но мы можем менять его с помощью режима визуализации **D3DRS_STENCILREF**. Например, приведенный ниже фрагмент кода сделает эталонное значение трафарета равным единице:

```
Device->SetRenderState(D3DRS_STENCILREF, 0x1);
```

Обратите внимание, что мы предпочитаем использовать шестнадцатеричную запись чисел, поскольку она позволяет сразу увидеть распределение битов в числе, а это очень полезно при выполнении поразрядных операций, таких как И.

8.1.3.2 Маска трафарета

Значение маски трафарета **mask** используется для маскирования (скрытия) отдельных разрядов в эталонном значении трафарета **ref** и значении из буфера трафарета **value**. По умолчанию значение маски равно **0xffffffff** и никакие разряды не маскируются. Можно изменить значение маски, установив состояние визуализации **D3DRS_STENCILMASK**. В приведенном ниже коде мы задаем значение, которое будет маскировать 16 старших разрядов:

```
Device->SetRenderState(D3DRS_STENCILMASK, 0x0000ffff);
```

ПРИМЕЧАНИЕ Если вы ничего не поняли из разговора о битах и масках, скорее всего вам надо почитать о двоичных и шестнадцатеричных числах, а также о поразрядных операциях.

8.1.3.3 Значение трафарета

Как упоминалось ранее это значение в буфере трафарета для текущего пикселя, который участвует в проверке трафарета. Например, если мы проводим проверку

трафарета для пикселя, находящегося в позиции (i, j), то значением будет число, хранящееся в позиции (i, j) буфера трафарета. Мы не можем явно устанавливать значения отдельных элементов буфера трафарета, но помните, что можно очищать буфер. Кроме того можно использовать режимы визуализации трафарета чтобы управлять тем, какие значения будут записаны в буфер. Относящиеся к работе с трафаретами режимы визуализации будут рассмотрены ниже.

8.1.3.4 Операция сравнения

Мы можем задать используемую операцию сравнения, установив режим визуализации **D3DRS_STENCILFUNC**. Операция сравнения должна быть членом перечисления **D3DCMPFUNC**:

```
typedef enum _D3DCMPFUNC {
     D3DCMP_NEVER = 1,
     D3DCMP_LESS = 2,
     D3DCMP_EQUAL = 3,
     D3DCMP_LESSEQUAL = 4,
     D3DCMP_GREATER = 5,
     D3DCMP_NOTEQUAL = 6,
     D3DCMP_GREATEREQUAL = 7,
     D3DCMP_ALWAYS = 8,
     D3DCMP_FORCE_DWORD = 0x7fffffff
} D3DCMPFUNC;
```

- **D3DCMP_NEVER** — Проверка трафарета всегда завершается неудачно.
- **D3DCMP_LESS** — Проверка трафарета завершается успешно, если $LHS < RHS$.
- **D3DCMP_EQUAL** — Проверка трафарета завершается успешно, если $LHS = RHS$.
- **D3DCMP_LESSEQUAL** — Проверка трафарета завершается успешно, если $LHS \leq RHS$.
- **D3DCMP_GREATER** — Проверка трафарета завершается успешно, если $LHS > RHS$.
- **D3DCMP_NOTEQUAL** — Проверка трафарета завершается успешно, если $LHS \neq RHS$.
- **D3DCMP_GREATEREQUAL** — Проверка трафарета завершается успешно, если $LHS \geq RHS$.
- **D3DCMP_ALWAYS** — Проверка трафарета всегда завершается успешно.

8.1.4 Обновление буфера трафарета

Помимо алгоритма принятия решения записывать конкретный пиксель во вторичный буфер или нет, мы можем задать правила обновления элементов буфера трафарета в следующих случаях:

- Проверка трафарета для пикселя в позиции (i, j) завершилась неудачно. Мы можем определить, как в таком случае будет обновляться элемент (i, j) буфера трафарета установив режим визуализации **D3DRS_STENCILFAIL**:

```
Device->SetRenderState(D3DRS_STENCILFAIL, StencilOperation);
```

- Тест глубины для пикселя в позиции (i, j) завершился неудачно. Мы можем определить, как в таком случае будет обновляться элемент (i, j) буфера трафарета установив режим визуализации **D3DRS_STENCILZFAIL**:

```
Device->SetRenderState(D3DRS_STENCILZFAIL, StencilOperation);
```

- Тест глубины и проверка трафарета для пикселя в позиции (i, j) завершились успешно. Мы можем определить, как в таком случае будет обновляться элемент (i, j) буфера трафарета установив режим визуализации **D3DRS_STENCILPASS**:

```
Device->SetRenderState(D3DRS_STENCILPASS, StencilOperation);
```

В приведенных выше примерах **StencilOperation** — это одна из перечисленных ниже предопределенных констант:

- **D3DSTENCILOP_KEEP** — Значение в буфере трафарета не должно меняться (следовательно, остается то значение, которое было в буфере до этого).
- **D3DSTENCILOP_ZERO** — Элементу буфера трафарета присваивается ноль.
- **D3DSTENCILOP_REPLACE** — Элемент буфера трафарета будет замен на эталонное значение трафарета.
- **D3DSTENCILOP_INCRSAT** — Элемент буфера трафарета будет увеличен. Если в результате увеличения будет превышено максимальное допустимое значение элемента буфера трафарета, элементу будет присвоено максимальное допустимое значение.
- **D3DSTENCILOP_DECRSAT** — Элемент буфера трафарета будет уменьшен. Если в результате уменьшения значение элемента буфера трафарета станет меньше нуля, элементу будет присвоен ноль.
- **D3DSTENCILOP_INVERT** — Элемент буфера трафарета будет поразрядно инвертирован.
- **D3DSTENCILOP_INCR** — Элемент буфера трафарета будет увеличен. Если в результате увеличения будет превышено максимальное допустимое значение элемента буфера трафарета, элементу будет присвоен ноль.
- **D3DSTENCILOP_DECR** — Элемент буфера трафарета будет уменьшен. Если в результате уменьшения значение элемента буфера трафарета

станет меньше нуля, элементу будет присвоено максимальное допустимое значение.

8.1.5 Маска записи трафарета

Помимо уже рассмотренных относящихся к трафарету режимов визуализации, можно устанавливать маску записи, которая будет маскировать заданные разряды в любом записываемом в буфер трафарета значении. Маска записи задается в режиме визуализации **D3DRS_STENCILWRITEMASK**. Значение по умолчанию — **0xffffffff**. В приведенном ниже примере устанавливается маска, которая обнуляет старшие 16 разрядов:

```
Device->SetRenderState(D3DRS_STENCILWRITEMASK, 0x0000ffff);
```

8.2 Пример приложения: зеркала

Многие поверхности в природе служат как зеркала и позволяют видеть отражающиеся в них объекты. В этом разделе будет показано, как можно имитировать зеркала в наших приложениях. Обратите внимание, что для упрощения задачи мы ограничимся только реализацией плоских зеркал. Полированная поверхность кузова автомобиля, к примеру, тоже отражает предметы, но она скругленная а не плоская. Вместо этого мы научимся визуализировать отражения предметов в отполированном мраморном полу или неоднократно виденные отражения в стенных зеркалах, иными словами, отражения в любых плоских поверхностях.

Программная реализация зеркал требует решения двух задач. Во-первых, надо узнать как формируется отражение объекта относительно заданной плоскости, чтобы мы могли правильно нарисовать его. Во-вторых, мы должны сделать так, чтобы отражение показывалось только в зеркале. Следовательно, мы должны как-то «отметить» поверхность зеркала, чтобы потом при визуализации рисовать отражение объекта только в том случае, если оно находится в зеркале. Эта концепция иллюстрируется на рис. 8.1.

Первая задача легко решается с помощью векторной геометрии. Вторую задачу можно решить с помощью буфера трафарета. В двух следующих разделах по отдельности разбирается решение этих двух задач. В третьем разделе эти две задачи будут объединены и мы рассиотрим код первого примера программы из данной главы — StencilMirrors.

8.2.1 Математика отражений

Сейчас мы посмотрим как вычислить точку $\mathbf{v}' = (v'_x, v'_y, v'_z)$, являющуюся отражением точки $\mathbf{v} = (v_x, v_y, v_z)$ относительно заданной плоскости $\hat{\mathbf{n}} \cdot \mathbf{p} + d = 0$. Во время обсуждения смотрите на рис. 8.2.

Из раздела «Плоскости» первой части книги мы знаем, что $\mathbf{q} = \mathbf{v} - k\hat{\mathbf{n}}$, где k — это кратчайшее расстояние от точки $\mathbf{v}$ до плоскости. Отсюда следует, что отражение точки $\mathbf{v}$ относительно плоскости ($\hat{\mathbf{n}}$, d) вычисляется следующим образом:

$$\begin{aligned}\mathbf{v}' &= \mathbf{v} - 2k\hat{\mathbf{n}} \\ &= \mathbf{v} - 2(\hat{\mathbf{n}} \cdot \mathbf{v} + d)\hat{\mathbf{n}} \\ &= \mathbf{v} - 2\left[(\hat{\mathbf{n}} \cdot \mathbf{v})\hat{\mathbf{n}} + d\hat{\mathbf{n}}\right]\end{aligned}$$

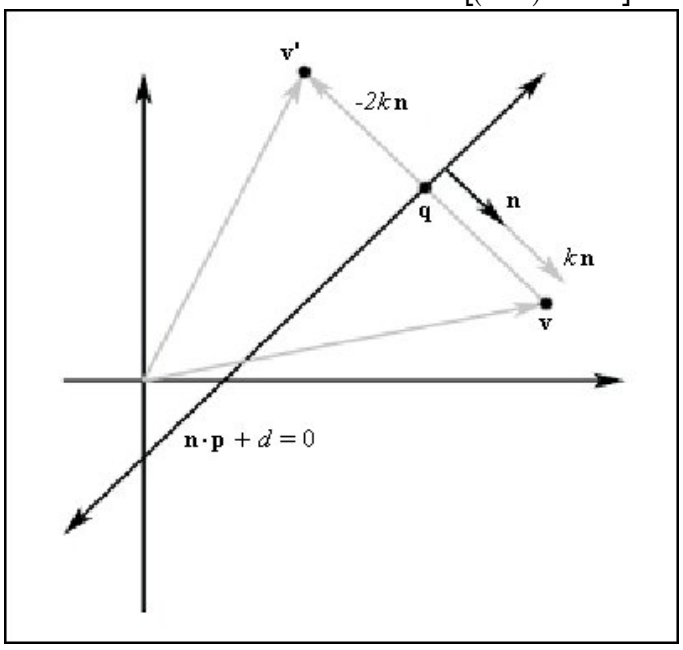

Рис. 8.2.*Отражение относительно заданной плоскости. Обратите внимание, что k — это кратчайшее расстояние от точки **v** до плоскости и на данном рисунке k положительно потому что точка **v** находится в положительном полупространстве плоскости*

Мы можем представить это преобразование точки $\mathbf{v}$ в точку $\mathbf{v}'$ с помощью следующей матрицы:

$$\mathbf{R} = \begin{bmatrix} -2n_x n_x + 1 & -2n_y n_x & -2n_z n_x & 0 \\ -2n_x n_y & -2n_y n_{y+} + 1 & -2n_z n_y & 0 \\ -2n_x n_z & -2n_y n_z & -2n_z n_z + 1 & 0 \\ -2n_x d & -2n_y d & -2n_z d & 1 \end{bmatrix}$$

Для создания матрицы отражения относительно заданной плоскости, обозначаемой $\mathbf{R}$, библиотека D3DX предоставляет следующую функцию:

```
D3DXMATRIX *D3DXMatrixReflect(
     D3DXMATRIX *pOut,       // Полученная матрица отражения
     CONST D3DXPLANE *pPlane // Плоскость, относительно которой
```

```
                                    // формируется отражение
);
```

Раз уж мы начали разбирать тему преобразования отражения, позвольте представить вам матрицы преобразования отражения для трех особых случаев. Это отражения относительно трех стандартных плоскостей координатной системы — плоскости YZ, плоскости XZ и плоскости XY — представляемые следующими тремя матрицами соответственно:

$$\mathbf{R}_{yz} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R}_{xz} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R}_{xy} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

При отражении точки относительно плоскости YZ мы просто меняем знак компоненты x. Аналогичным образом, чтобы отразить точку относительно плоскости XZ, мы меняем знак компоненты y. И, наконец, при отражении точки относительно плоскости XY мы меняем знак компоненты z. Эти отражения легко заметить, наблюдая симметрию относительно каждой из стандартных плоскостей координатной системы.

8.2.2 Обзор реализации отражений

При реализации зеркал необходимо, чтобы объект отражался только в том случае, если находится перед зеркалом. Однако, мы не хотим выполнять проверку положения в пространстве находящихся перед зеркалом объектов, поскольку она очень сложна. Вместо этого мы всегда вычисляем и визуализируем отражения объектов, независимо от их местоположения. В результате такого упрощения возникает проблема, показанная на рис. 8.1 в начале главы. Как видите, отражение объекта (в данном случае чайника) отображается и на тех поверхностях, которые не являются зеркальными (например, на стенах). Эту проблему можно решить с помощью использования буфера трафарета, поскольку буфер трафарета позволяет блокировать визуализацию для заданных областей вторичного буфера. Следовательно, мы можем использовать буфер трафарета чтобы блокировать визуализацию тех частей отражения чайника, которые находятся вне поверхности зеркала. Ниже приведен краткий обзор действий, которые необходимо выполнить для корректной визуализации отражений.

1. Визуализируем всю сцену (пол, стены, зеркало и чайник) как обычно. Отражение чайника на данном этапе *не визуализируется*. Обратите внимание, что на этом этапе содержимое буфера трафарета не меняется.
2. Очишаем буфер трафарета, заполняя его нулями. На рис. 8.3. показано состояние вторичного буфера и буфера трафарета на данном этапе.

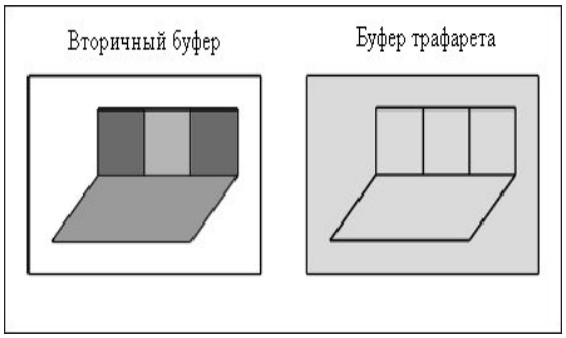

Рис. 8.3. *Сцена, визуализированная во вторичный буфер, и заполненный нулями буфер трафарета. Светло-серым цветом отмечены заполненные нулями пиксели буфера трафарета*

3. Визуализируем образующие зеркало примитивы *только в буфер трафарета*. Указываем, что проверка трафарета должна всегда завершаться успешно, а элемент буфера трафарета при успешном завершении проверки должен быть заменен на 1. Поскольку мы визуализируем только зеркало, всем пикселям буфера трафарета будет присвоено нулевое значение, за исключением тех пикселей, которые соответствуют изображению зеркала — им будет присвоено значение 1. Обновленный буфер трафарета показан на рис. 8.4. Итак, мы отметили пиксели зеркала в буфере трафарета.

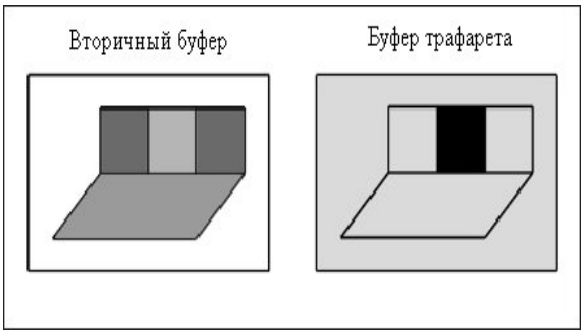

Рис. 8.4. *Визуализация зеркала в буфер трафарета, предназначенная для отметки тех пикселей, которые соответствуют изображению зеркала. Черная область буфера трафарета соответствует пикселям, которым присвоено значение 1*

4. Теперь мы визуализируем отражение чайника во вторичный буфер и буфер трафарета. Вспомните, что изображение во вторичный буфер визуализируется только если пройдена проверка трафарета. Сейчас нам надо задать такую проверку трафарета, которая будет завершаться успешно, только если в буфере трафарета для данного пикселя записана 1. В результате изображение чайника будет отображаться только в том случае, если в соответствующем пикселю элементе буфера трафарета записана единица. Поскольку единицы в буфере трафарета записаны только в тех элементах, которые соответствуют пикселям зеркала, отражение чайника будет показано только в зеркале.

8.2.3 Код и комментарии

Относящийся к данному примеру код находится в функции **`RenderMirror`**, которая сперва визуализирует примитивы зеркала в буфер трафарета, а затем

отображает отражение чайника, но только в том случае, если оно находится в зеркале. Мы исследуем функцию **RenderMirror** строка за строкой и обсудим что и, самое главное, зачем она делает.

Если в качестве путеводителя вы используете список этапов, приведенный в разделе 8.2.2, обратите внимание, что мы начинаем обсуждение с этапа 3, поскольку ни первый ни второй этап не влияют на буфер трафарета. И учтите, что в следующем ниже объяснении мы будем обсуждать только визуализацию зеркал.

Обратите внимание, что обсуждение кода разделено на несколько частей - это сделано только для того, чтобы сделать дискуссию более модульной.

8.2.3.1 Часть I

Мы начинаем с разрешения использования буфера трафарета и установки связанных с ним режимов визуализации:

```
void RenderMirror()
{
Device->SetRenderState(D3DRS_STENCILENABLE,    true);
Device->SetRenderState(D3DRS_STENCILFUNC,      D3DCMP_ALWAYS);
Device->SetRenderState(D3DRS_STENCILREF,       0x1);
Device->SetRenderState(D3DRS_STENCILMASK,      0xffffffff);
Device->SetRenderState(D3DRS_STENCILWRITEMASK, 0xffffffff);
Device->SetRenderState(D3DRS_STENCILZFAIL,     D3DSTENCILOP_KEEP);
Device->SetRenderState(D3DRS_STENCILFAIL,      D3DSTENCILOP_KEEP);
Device->SetRenderState(D3DRS_STENCILPASS,      D3DSTENCILOP_REPLACE);
```

Все действия исключительно прямолинейны. Мы устанавливаем для трафарета операцию сравнения **D3DCMP_ALWAYS**, чтобы проверка трафарета всегда завершалась успешно.

Мы устанавливаем режим обновления буфера трафарета **D3DSTENCILOP_KEEP**, чтобы в том случае, если тест глубины не пройден, соответствующий элемент буфера трафарета не менялся. Таким образом, мы будем сохранять его текущее значение. Мы делаем это потому, что если тест глубины не пройден, значит соответствующий пиксель скрыт и нам не надо визуализировать ту часть отражения, которая попадает на невидимые пиксели.

Мы также задаем режим обновления буфера трафарета **D3DSTENCILOP_KEEP** и для того случая, когда не пройдена проверка трафарета. Здесь в этом нет никакой необходимости, ведь проверка трафарета всегда завершается успешно, поскольку мы задали для нее режим **D3DCMP_ALWAYS**. Однако чуть позже мы изменим операцию сравнения и нам потребуется задать параметр, определяющий поведение системы в случае неудачного завершения проверки трафарета; так что лучше сделать это сразу.

Для того случая, когда пройдены тест глубины и проверка трафарета, мы задаем режим обновления **D3DSTENCILOP_REPLACE**, указывающий, что элементы буфера трафарета будут заменяться на эталонное значение трафарета — **0x1**.

8.2.3.2 Часть II

В следующей части кода выполняется визуализация зеркала, но только в буфер трафарета. Чтобы запретить запись в буфер глубины мы присваиваем режиму визуализации **D3DRS_ZWRITEENABLE** значение **false**. Чтобы заблокировать изменение вторичного буфера, мы присваиваем коэффициенту смешивания источника значение **D3DBLEND_ZERO**, а коэффициенту смешивания приемника — значение **D3DBLEND_ONE**. Подставив эти коэффициенты в формулу смешивания мы увидим, что вторичный буфер остается неизменным:

ИтоговыйПиксель =
= *ПиксельИсточника* ⊗ (0, 0, 0, 0) + *ПиксельПриемника* ⊗ (1, 1, 1, 1) =
= (0, 0, 0, 0) + *ПиксельПриемника* = *ПиксельПриемника*

```
// Запрещаем запись во вторичный буфер и буфер глубины
Device->SetRenderState(D3DRS_ZWRITEENABLE, false);
Device->SetRenderState(D3DRS_ALPHABLENDENABLE, true);
Device->SetRenderState(D3DRS_SRCBLEND,  D3DBLEND_ZERO);
Device->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);

// Рисуем зеркало в буфере трафарета
Device->SetStreamSource(0, VB, 0, sizeof(Vertex));
Device->SetFVF(Vertex::FVF);
Device->SetMaterial(&MirrorMtrl);
Device->SetTexture(0, MirrorTex);
D3DXMATRIX I;
D3DXMatrixIdentity(&I);
Device->SetTransform(D3DTS_WORLD, &I);
Device->DrawPrimitive(D3DPT_TRIANGLELIST, 18, 2);

// Разрешаем запись в буфер глубины
Device->SetRenderState(D3DRS_ZWRITEENABLE, true);
```

8.2.3.3 Часть III

К этому моменту тем пикселям буфера трафарета, которые соответствуют видимым пикселям зеркала, присвоено значение **0x1**, так что у нас отмечена область в которой будет рисоваться отражение. Сейчас мы будем готовиться к визуализации отражения чайника. Вспомните, что визуализировать отражение надо только в тех пикселях, которые соответствуют зеркалу. Это легко сделать, поскольку необходимые пиксели уже отмечены в буфере трафарета.

Мы устанавливаем следующие режимы визуализации:

```
Device->SetRenderState(D3DRS_STENCILFUNC, D3DCMP_EQUAL);
Device->SetRenderState(D3DRS_STENCILPASS, D3DSTENCILOP_KEEP);
```

Задав новую операцию сравнения мы получаем следующую проверку трафарета:

```
       (ref & mask == (value & mask)
(0x1 & 0xffffffff) == (value & 0xffffffff)
            (0x1) == (value & 0xffffffff)
```

Это означает, что проверка трафарета будет пройдена успешно только в том случае, если **value = 0x1**. Поскольку значение **0x1** записано только в тех частях буфера трафарета, которые соответствуют изображению зеркала, проверка будет успешно пройдена только когда выполняется визуализация в этих областях. Следовательно, отражение чайника будет нарисовано только в зеркале и не будет рисоваться на других поверхностях.

Обратите внимание, что мы меняем значение режима визуализации **D3DRS_STENCILPASS** на **D3DSTENCILOP_KEEP**, чтобы в случае успешного прохождения проверки значение в буфере трафарета не менялось. Следовательно, в последующих проходах визуализации значения в буфере трафарета останутся неизменными (это задано значением **D3DSTENCILOP_KEEP**). Мы используем буфер трафарета только для отметки тех пикселов, которые соответствуют изображению зеркала.

8.2.3.4 Часть IV

В следующей части функции **RenderMirror** производится вычисление матрицы, которая размещает отражение в сцене:

```
// Размещение отражения
D3DXMATRIX W, T, R;
D3DXPLANE plane(0.0f, 0.0f, 1.0f, 0.0f); // плоскость XY
D3DXMatrixReflect(&R, &plane);

D3DXMatrixTranslation(&T,
            TeapotPosition.x,
            TeapotPosition.y,
            TeapotPosition.z);

W = T * R;
```

Обратите внимание, что мы сперва переносим отражение в то место, где находится исходный чайник. После этого мы выполняем отражение относительно плоскости XY. Порядок выполнения преобразований определяется порядком следования матриц в операции умножения.

8.2.3.5 Часть V

Мы почти готовы к визуализации отражения чайника. Однако, если выполнить визуализацию сейчас, никакого отражения не появится. Почему? Потому что значения глубины для отражения чайника больше, чем значения глубины пикселей зеркала и, с технической точки зрения, примитивы зеркала закрывают отражение чайника. Чтобы обойти эту проблему мы очищаем буфер глубины:

```
Device->Clear(0, 0, D3DCLEAR_ZBUFFER, 0, 1.0f, 0);
```

Тем не менее, пока решены не все проблемы. Если просто очистить буфер глубины, отражение чайника будет нарисовано поверх зеркала и будет выглядеть неестественно. Необходимо не только очистить буфер глубины, но и выполнить смешивание пикселей отражения с пикселями зеркала. Благодаря этому

отражение будет выглядеть так, будто оно находится в глубине зеркала. Для смешивания пикселей отражения чайника с пикселями зеркала мы будем использовать следующую формулу:

ИтоговыйПиксель =
= *ПиксельИсточника* ⊗ *ПиксельПриемника* + *ПиксельПриемника* ⊗ (0, 0, 0, 0) =
= *ПиксельИсточника* ⊗ *ПиксельПриемника*

Пиксель источника берется из отражения чайника, а писксель приемника — из зеркала, и формула помогает нам увидеть как они соединяются воедино. В коде это выглядит так:

```
Device->SetRenderState(D3DRS_SRCBLEND,  D3DBLEND_DESTCOLOR);
Device->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ZERO);
```

Наконец-то мы готовы к рисованию отражения чайника:

```
Device->SetTransform(D3DTS_WORLD, &W);
Device->SetMaterial(&TeapotMtrl);
Device->SetTexture(0, 0);

Device->SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);
Teapot->DrawSubset(0);
```

Вспомните, что в разделе 8.2.3.4 мы получили матрицу **W**, которая помещает отражение чайника в предназначенное ему место сцены. Также обратите внимание, что мы меняем режим удаления невидимых граней. Это необходимо потому, что при отражении объекта его фронтальные и обратные полигоны меняются местами; при этом порядок обхода вершин не меняется. Таким образом порядок обхода вершин «новых» фронтальных граней будет указывать Direct3D, что они являются обратными полигонами. Аналогичным образом порядок обхода вершин «новых» обратных граней будет убеждать Direct3D в том, что они являются фронтальными. Следовательно, для коррекции нам следует изменить условие отбрасывания обратных граней.

Возвращая все в исходное состояние мы запрещаем смешивание и работу буфера трафарета, после чего восстанавливаем стандартный режим отбрасывания обратных граней:

```
Device->SetRenderState(D3DRS_ALPHABLENDENABLE, false);
Device->SetRenderState(D3DRS_STENCILENABLE,    false);
Device->SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);

} // конец функции RenderMirror()
```

8.3 Пример приложения: плоская тень

Тени помогают нашему восприятию определить откуда на сцену падает свет и являются незаменимым инструментом для добавления сценам реализма. В данном разделе мы покажем как реализуются плоские тени — то есть такие тени, которые отбрасываются на плоскую поверхность (рис. 8.5).

Следует отметить, что тени данного типа являются трюком, и хотя улучшают вид сцены, выглядят не так реалистично, как теневые объемы. Теневые объемы (shadow volumes) являются достаточно сложной темой и мы решили, что не стоит рассматривать их в книге начального уровня. Однако стоит помнить, что в DirectX SDK есть пример программы, демонстрирующей теневые объемы.

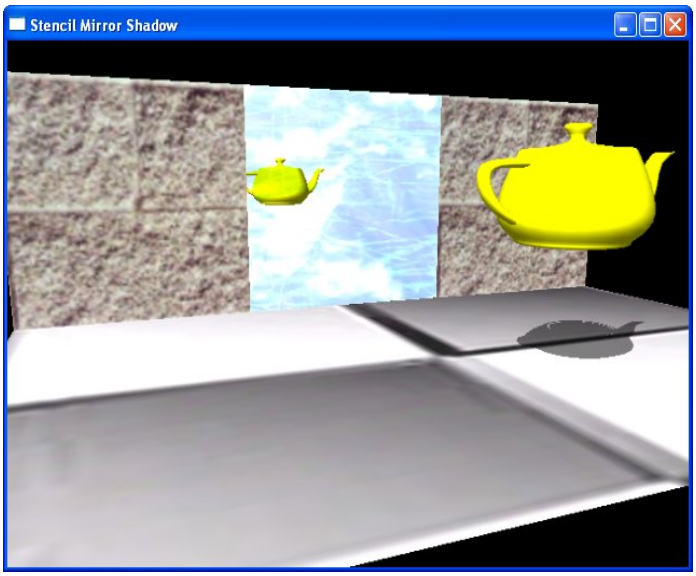

Рис. 8.5. *Окно рассматриваемого в этом разделе приложения. Обратите внимание на отбрасываемую чайником тень на полу*

Чтобы реализовать плоские тени мы должны сперва определить тень, которую объект отбрасывает на плоскость, и смоделировать ее геометрически чтобы потом можно было визуализировать ее. Первая часть легко решается с помощью трехмерной математики. Затем мы визуализируем образующие тень полигоны задав для них черный материал и 50% прозрачность. Визуализация теней может привести к возникновению артефактов, называемых «двойным смешиванием», о которых мы поговорим в последующих разделах. Чтобы предотвратить возникновение двойного смешивания мы задействуем буфер трафарета.

8.3.1 Тени от параллельного источника света

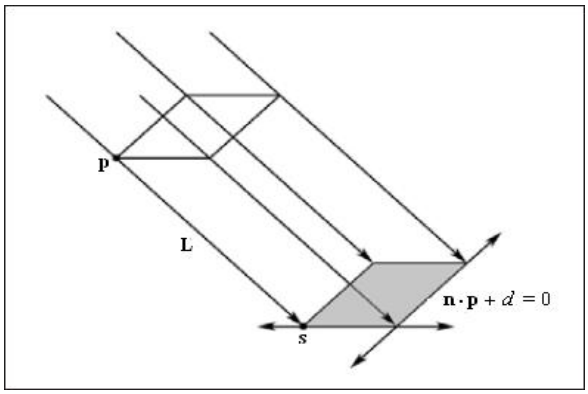

Рис. 8.6. *Тень, отбрасываемая объектом при его освещении параллельным источником света*

На рис. 8.6 показана тень, отбрасываемая объектом при его освещении параллельным источником света. Луч света от параллельного источника, падающий в направлении **L**, и проходящий черз вершину **p** описывается формулой $\mathbf{r}(t) = \mathbf{p} + t\mathbf{L}$. Пересечение луча $\mathbf{r}(t)$ с плоскостью $\mathbf{n} \cdot \mathbf{p} + d = 0$ дает точку **s**. Набор точек пересечения, определяемый путем вычисления пересечения лучей $\mathbf{r}(t)$, проходящих через каждую из вершин объекта, с плоскостью, задает геометрию тени. Точка пересечения **s** легко вычисляется с помощью формулы проверки пересечения луча и плоскости:

$$\mathbf{n} \cdot (\mathbf{p} + t\mathbf{L}) + d = \mathbf{0}$$

Подставляем $\mathbf{r}(t)$ в формулу плоскости $\mathbf{n} \cdot \mathbf{p} + d = 0$.

$$\mathbf{n} \cdot \mathbf{p} + t(\mathbf{n} \cdot \mathbf{L}) = -d$$

$$t(\mathbf{n} \cdot \mathbf{L}) = -d - \mathbf{n} \cdot \mathbf{p}$$

$$t = \frac{-d - \mathbf{n} \cdot \mathbf{p}}{\mathbf{n} \cdot \mathbf{L}}$$

Решение для t.

Тогда:

$$\mathbf{s} = \mathbf{p} + \left[\frac{-d - \mathbf{n} \cdot \mathbf{p}}{\mathbf{n} \cdot \mathbf{L}} \right] \mathbf{L}$$

8.3.2 Тени от точечного источника света

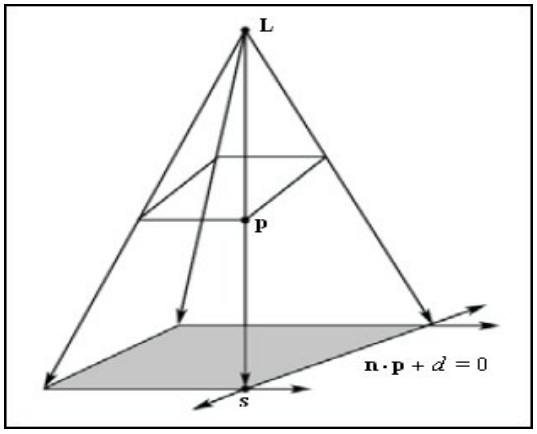

Рис. 8.7. *Тень, отбрасываемая объектом при его освещении точечным источником света*

На рис. 8.7 показана тень, отбрасываемая объектом при его освещении точечным источником света, находящимся в точке **L**. Лучи света от точечного источника света, проходящие через вершину **p** описываются формулой $\mathbf{r}(t) = \mathbf{p} + t(\mathbf{p} - \mathbf{L})$. Пересечение луча $\mathbf{r}(t)$ с плоскостью $\mathbf{n} \cdot \mathbf{p} + d = 0$ дает точку **s**. Набор точек пересечения, определяемый путем вычисления пересечения лучей $\mathbf{r}(t)$, проходящих через каждую из вершин объекта, с плоскостью, задает геометрию тени. Точка **s** определяется с помощью той же самой техники (формулы пересечения луча и плоскости), которую мы уже рассмотрели в разделе 8.3.1.

ПРИМЕЧАНИЕ Обратите внимание, что для точечного и параллельного света вектор **L** используется различным образом. В случае точечного источника света **L** определяет местоположение источника света. В случае параллельного источника света **L** определяет направление падающих лучей.

8.3.3 Матрица тени

Из рис. 8.6 следует, что для параллельного источника света тень получается путем *параллельной* проекции объекта на плоскость $\mathbf{n} \cdot \mathbf{p} + d = 0$ в направлении вектора распространения лучей света. Аналогичным образом, рис. 8.7 показывает, что для точечного света тень получается путем *перспективной* проекции объекта на плоскость $\mathbf{n} \cdot \mathbf{p} + d = 0$ с центром проекции, находящимся в той же точке, что и источник света.

Мы можем представить преобразование вершины **p** в ее проекцию **s** на плоскость $\mathbf{n} \cdot \mathbf{p} + d = 0$ в виде матрицы. Более того, проявив некоторую изобретательность мы можем представить параллельную и перспективную проекцию с помощью одной матрицы.

Пусть (n_x, n_y, n_z, d) — это четырехмерный вектор, представляющий коэффициенты обобщенной формулы плоскости, описывающие плоскость на которую отбрасывается тень. Пусть $\mathbf{L} = (L_x, L_y, L_z, L_w)$ — это четырехмерный

вектор, описывающий либо направление лучей параллельного источника света, либо местоположение точечного источника света. Для определения типа источника света мы будем использовать компоненту w следующим образом:

1. Если $w = 0$, то **L** описывает направление лучей параллельного источника света.
2. Если $w = 1$, то **L** описывает местоположение точечного источника света.

Предполагая, что вектор нормали плоскости нормализован, мы получим $k = (n_x, n_y, n_z, d) \cdot (L_x, L_y, L_z, L_w) = n_xL_x + n_yL_y + n_zL_z + dL_w$. Тогда мы можем представить преобразование вершины **p** в ее проекцию **s** в виде следующей *матрицы тени* (*shadow matrix*):

$$\mathbf{S} = \begin{bmatrix} n_xL_x + k & n_xL_y & n_xL_z & n_xL_w \\ n_yL_x & n_yL_y + k & n_yL_z & n_yL_w \\ n_zL_x & n_zL_y & n_zL_z + k & n_zL_w \\ dL_x & dL_y & dL_z & dL_w + k \end{bmatrix}$$

Мы не будем показывать получение этой матрицы, поскольку она больше нигде не будет использоваться и информация о ее получении не представляет особой важности. Тем не менее, интересующихся читателей мы отсылаем к шестой главе книги «Jim Blinn's Corner: A Trip Down the Graphics Pipeline», где показано как получена эта матрица.

Библиотека D3DX предоставляет следующую функцию для получения матрицы тени по данным плоскости, на которую отбрасывается тень, и вектору, описывающему направление лучей параллельного источника света, если компонента $w = 0$ или местоположение точечного источника света, если компонента $w = 1$:

```
D3DXMATRIX *D3DXMatrixShadow(
     D3DXMATRIX        *pOut,
     CONST D3DXVECTOR4 *pLight, // L
     CONST D3DXPLANE   *pPlane  // плоскость, на которую
                                // отбрасывается тень
);
```

8.3.4 Использование буфера трафарета для предотвращения двойного смешивания

Когда мы проецируем объект на плоскость для описания тени может получиться так, что два или более спроецированных треугольника будут перекрываться. Когда мы будем визуализировать прозрачную тень (используя смешивание), для тех областей, где треугольники перекрываются, смешивание будет выполнено несколько раз, и, в результате они будут более темными (рис. 8.8).

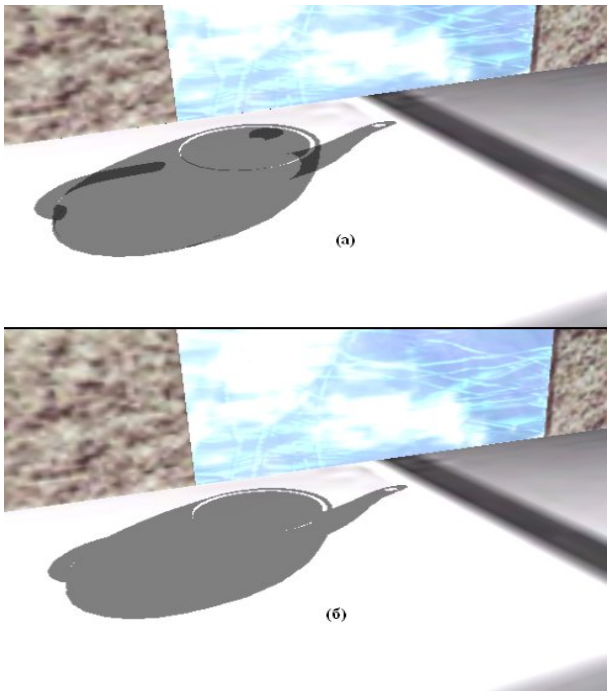

Рис. 8.8. *Обратите внимание на темные области тени на рисунке (а). Они соответствуют тем частям проекции в которых фрагменты изображения перекрываются, что приводит к «двойному смешиванию». На рисунке (б) показана правильно визуализированная тень без двойного смешивания*

Мы можем решить эту проблему с помощью буфера трафарета. Мы устанавливаем проверку трафарета таким образом, чтобы ее проходили только те пиксели, которые визуализируются в первый раз. Таким образом, когда мы визуализируем пиксели тени во вторичный буфер, то одновременно устанавливаем соответствующие элементы буфера трафарета. Если мы потом попытаемся записать новый пиксель в ту область, куда уже был визуализирован пиксель тени (которая была отмечена в буфере трафарета), проверка трафарета не будет пройдена. Так мы предотвратим запись перекрывающихся пикселей и, следовательно, устраним артефакты, вызываемые двойным смешиванием.

8.3.5 Код и комментарии

В этом разделе мы исследуем код, взятый из примера Shadow, находящегося в сопроводительных файлах к этой книге. Код, который важен для данного примера, находится в функции **`RenderShadow`**. Обратите внимание, что мы подразумеваем, что буфер трафарета уже очищен и заполнен нулями.

Мы начинаем с установки относящихся к работе трафарета режимов визуализации. Мы устанавливаем функцию сравнения **`D3DCMP_EQUAL`**, а режим визуализации **`D3DRS_STENCILREF`** равным **`0x0`**, в результате чего пиксели тени будут визуализироваться во вторичный буфер если значение соответствующего элемента буфера трафарета равно **`0x0`**.

Поскольку буфер трафарета очищен и заполнен нулями (**`0x0`**), при первой записи пикселя тени проверка будет всегда звершаться успешно; но поскольку мы присвоили режиму **`D3DRS_STENCILPASS`** значение **`D3DSTENCILOP_INCR`**,

если мы еще раз попытаемся записать пиксель в то же самое место, проверка трафарета не будет пройдена. Когда мы записываем самый первый пиксель тени, соответствующее ему значение в буфере трафарета увеличивается и страновится равным **0x1**, после этого при любой другой попытке записать пиксель в то же самое место проверка трафарета не будет пройдена. Так мы праедотвращаем перезапись пикселей и двойное смешивание.

```
void RenderShadow()
{
     Device->SetRenderState(D3DRS_STENCILENABLE,    true);
     Device->SetRenderState(D3DRS_STENCILFUNC,      D3DCMP_EQUAL);
     Device->SetRenderState(D3DRS_STENCILREF,       0x0);
     Device->SetRenderState(D3DRS_STENCILMASK,      0xffffffff);
     Device->SetRenderState(D3DRS_STENCILWRITEMASK, 0xffffffff);
     Device->SetRenderState(D3DRS_STENCILZFAIL,   D3DSTENCILOP_KEEP);
     Device->SetRenderState(D3DRS_STENCILFAIL,    D3DSTENCILOP_KEEP);
     Device->SetRenderState(D3DRS_STENCILPASS,    D3DSTENCILOP_INCR);
```

Затем мы вычисляем матрицу преобразования тени и перемещаем тень в требуемое место сцены.

```
     // Вычисление преобразования проекции чайника
     // в тень
     D3DXVECTOR4 lightDirection(0.707f, -0.707f, 0.707f, 0.0f);
     D3DXPLANE groundPlane(0.0f, -1.0f, 0.0f, 0.0f);

     D3DXMATRIX S;
     D3DXMatrixShadow(&S, &lightDirection, &groundPlane);

     D3DXMATRIX T;
     D3DXMatrixTranslation(&T, TeapotPosition.x, TeapotPosition.y,
                           TeapotPosition.z);

     D3DXMATRIX W = T * S;
     Device->SetTransform(D3DTS_WORLD, &W);
```

После этих действий мы устанавливаем черный материал с 50% прозрачностью, запрещаем проверку глубины, визуализируем тень и возвращаем все к исходному состоянию, вновь включая буфер глубины и запрещая альфа-смешивание и проверку трафарета. Мы отключаем буфер глубины чтобы предотвратить *z-конфликты* (*z-fighting*), приводящие к возникновению артефактов изображения, когда в буфере глубины у двух различных поверхностей записано одинаковое значение глубины; механизм визуализации не может определить, какая поверхность должна располагаться поверх другой и может отображать то одну поверхность, то другую. Визуализируя сперва пол и только потом, после отключения проверки глубины, тень мы гарантируем, что тень будет нарисована поверх пола.

ПРИМЕЧАНИЕ Альтернативным методом предотвращения z-конфликтов является использование поддерживаемого Direct3D механизма *смещения выборки глубины* (*depth bias*). Для получения дополнительной информации посмотрите описания режимов визуализации

D3DRS_DEPTHBIAS и **D3DRS_SLOPESCALEDEPTHBIAS** в документации к SDK.

```
    Device->SetRenderState(D3DRS_ALPHABLENDENABLE, true);
    Device->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
    Device->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);

    D3DMATERIAL9 mtrl = d3d::InitMtrl(d3d::BLACK, d3d::BLACK,
                                      d3d::BLACK, d3d::BLACK, 0.0f);
    mtrl.Diffuse.a = 0.5f; // 50% прозрачность

    // Отключаем буфер глубины, чтобы предотвратить z-конфликты
    // при визуализации тени поверх пола
    Device->SetRenderState(D3DRS_ZENABLE, false);

    Device->SetMaterial(&mtrl);
    Device->SetTexture(0, 0);
    Teapot->DrawSubset(0);

    Device->SetRenderState(D3DRS_ZENABLE,          true);
    Device->SetRenderState(D3DRS_ALPHABLENDENABLE, false);
    Device->SetRenderState(D3DRS_STENCILENABLE,    false);
} // конец функции RenderShadow()
```

8.4 Итоги

- Буфер трафарета и буфер глубины совместно используют одну и ту же поверхность и, следовательно, создаются одновременно. Формат поверхности буферов глубины/трафарета задается путем указания одного из членов перечисления **D3DFORMAT**.
- Трафарет используется для того, чтобы блокировать растеризацию отдельных пикселей. Как мы видели в этой главе, данная возможность, помимо других применений, полезна при реализации зеркал и теней.
- Мы можем управлять работой и обновлением буфера трафарета с помощью режимов визуализации **D3DRS_STENCIL***.
- Вот еще несколько приложений, которые могут быть реализованы с использованием буфера трафарета:
    - Теневые объемы.
    - Наплыв и затемнение.
    - Визуализация сложных сцен.
    - Контуры и силуэты.
    - Конструирование твердых тел.
    - Предотвращение z-конфликтов, вызванных совпадением плоскостей различных объектов.

Часть III
Применение Direct3D

В этой части мы сосредоточимся на применении Direct3D для реализации ряда трехмерных приложений, демонстрирующих такие техники как визуализация ландшафтов, системы частиц, выбор объектов и создание гибкого класса трехмерной камеры. Кроме того, мы посвятим некоторое время продолжению изучения библиотеки D3DX (в частности, тем ее частям, которые относятся к работе с сетками). Вот краткий обзор составляющих эту часть глав.

Глава 9, «Шрифты» — Во время игры часто требуется отобразить предназначенную для пользователя текстовую информацию. В этой главе мы обсудим три способа вывода текста, поддерживаемых Direct3D.

Глава 10, «Сетки: часть I» — Эта глава посвящена детальному исследованию членов данных и методов интерфейса сеток **`ID3DXMesh`** из библиотеки D3DX.

Глава 11, «Сетки: часть II» — В этой главе мы продолжим изучение относящихся к сеткам интерфейсов и функций библиотеки D3DX. Мы узнаем о файлах .X и о том, как загружать и визуализировать их. Кроме того, мы изучим интерфейс прогрессивных сеток **`ID3DXPMesh`**. В главе также будет рассказано о том, как вычислить ограничивающий прямоугольник и ограничивающую сферу для сетки.

Глава 12, «Построение гибкого класса камеры» — В этой главе мы разработаем и реализуем гибкий класс камеры с шестью степенями свободы. Такая камера может использоваться в авиационных имитаторах и играх с видом от первого лица.

Глава 13, «Основы визуализации ландшафтов» — Глава показывает как создать, текстурировать, осветить и визуализировать трехмерный ландшафт. Помимо этого мы покажем как можно плавно перемещать камеру, чтобы создавалось впечатление ходьбы по созданному ландшафту.

Глава 14, «Системы частиц» — В этой главе мы узнаем о том, как моделировать системы, состоящие из большого количества мелких частиц, которые ведут себя одинаковым образом. Системы частиц могут использоваться, например, для моделирования падающих снега и дождя, вспышек взрывов, клубов дыма, следов от ракет и даже пуль.

Глава 15, «Выбор объектов» — Эта глава посвящена описанию алгоритма, позволяющего определить, какой именно трехмерный объект сцены выбрал пользователь с помощью мыши. Выбор объектов необходим для трехмерных игр и приложений в которых пользователь взаимодействует с трехмерным виртуальным окружением с помощью мыши.

Глава 9
Шрифты

В ходе игры нам часто требуется показать пользователю какую-либо текстовую информацию. В этой главе мы рассмотрим три способа отображения текста, поддерживаемые Direct3D. Каждый способ иллюстрируется примером программы, находящимся на сайте этой книги и в сопроводительных файлах.

Цели

- Узнать, как можно визуализировать текст с помощью интерфейса **`ID3DXFont`**.
- Узнать, как можно визуализировать текст с помощью класса **`CD3DFont`**.
- Узнать, как вычислить частоту кадров приложения.
- Узнать, как создать и визуализировать трехмерный текст с помощью функции **`D3DXCreateText`**.

9.1 ID3DXFont

Библиотека D3DX предоставляет интерфейс **ID3DXFont**, который можно использовать для отображения текста в Direct3D-приложениях. Этот интерфейс использует для отображения текста GDI и его применение может значительно уменьшить быстродействие приложения. Однако, поскольку **ID3DXFont** использует GDI, он поддерживает сложные шрифты и форматирование.

9.1.1 Создание ID3DXFont

Для создания интерфейса **ID3DXFont** может использоваться функция **D3DXCreateFontIndirect**.

```
HRESULT D3DXCreateFontIndirect(
     LPDIRECT3DDEVICE9 pDevice, // устройство, связанное со шрифтом

     CONST LOGFONT* pLogFont, // структура LOGFONT, описывающая шрифт
     LPD3DXFONT* ppFont       // возвращает созданный шрифт
);
```

Приведенный ниже фрагмент кода показывает использование этой функции:

```
LOGFONT lf;
ZeroMemory(&lf, sizeof(LOGFONT));
lf.lfHeight    = 25;    // в логических единицах
lf.lfWidth     = 12;    // в логических единицах
lf.lfWeight    = 500;   // насыщенность,
                        // диапазон 0(тонкий) - 1000(жирный)
lf.lfItalic    = false;
lf.lfUnderline = false;
lf.lfStrikeOut = false;
lf.lfCharSet   = DEFAULT_CHARSET;
strcpy(lf.lfFaceName, "Times New Roman"); // гарнитура шрифта

ID3DXFont* font = 0;
D3DXCreateFontIndirect(Device, &lf, &font);
```

Обратите внимание, что сперва требуется заполнить структуру **LOGFONT**, которая описывает параметры создаваемого шрифта.

ПРИМЕЧАНИЕ Для получения указателя на интерфейс **ID3DXFont** вы можете также воспользоваться функцией **D3DXCreateFont**.

9.1.2 Рисование текста

После того, как мы получили указатель на интерфейс **ID3DXFont**, рисование текста осуществляется простым вызовом метода **ID3DXFont::DrawText**.

```
INT ID3DXFont::DrawText(
     LPCSTR pString,
     INT Count,
     LPRECT pRect,
     DWORD Format,
     D3DCOLOR Color
);
```

- **pString** — Указатель на отображаемую строку текста.
- **Count** — Количество отображаемых символов строки. Если строка завершается нулевым символом можно указать –1, чтобы строка отображалась вся.
- **pRect** — Указатель на структуру **RECT**, определяющую область экрана в которой будет отображаться текст.
- **Format** — Необязательные флаги, определяющие форматирование выводимого текста; их описание находится в документации к SDK.
- **Color** — Цвет текста.

Вот пример использования метода:

```
Font->DrawText(
     "Hello World",     // Выводимая строка
     -1,                // Строка завершается нулевым символом
     &rect,             // Прямоугольная область для рисования строки
     DT_TOP | DT_LEFT, // Рисуем в верхнем левом углу области
     0xff000000);       // Черный цвет
```

9.1.3 Вычисление частоты кадров

Примеры приложений к этой главе ID3DXFont и CFont вычисляют и отображают количество визуализируемых за секунду кадров (FPS). В этом разделе мы покажем как вычисляется FPS.

Сперва мы объявляем три глобальных переменных:

```
DWORD FrameCnt;    // Количество выведенных кадров
float TimeElapsed; // Прошедшее время
float FPS;         // Частота визуализации кадров
```

Мы вычисляем FPS каждую секунду; это дает нам достоверное среднее значение. Кроме того, мы храним вычисленное значение частоты кадров в течение одной секунды, что дает достаточно времени, чтобы прочитать его перед очередным изменением.

Итак, каждый кадр мы увеличиваем значение переменной **FrameCnt** и прибавляем к переменной **TimeElapsed** время, прошедшее с вывода предыдущего кадра:

```
FrameCnt++;
TimeElapsed += timeDelta;
```

где **timeDelta** — это время, прошедшее между двумя кадрами.

После того, как пройдет одна секунда, мы вычисляем частоту кадров по следующей формуле:

```
FPS = (float)FrameCnt / TimeElapsed;
```

Затем мы обнуляем переменные **FrameCnt** и **TimeElapsed** и начинаем вычисление среднего значения частоты кадров для следующей секунды. Вот как выглядит весь код вместе:

```
void CalcFPS(float timeDelta)
{
     FrameCnt++;
     TimeElapsed += timeDelta;

     if(TimeElapsed >= 1.0f)
     {
          FPS = (float)FrameCnt / TimeElapsed;
          TimeElapsed = 0.0f;
          FrameCnt = 0;
     }
}
```

9.2 CD3DFont

DirectX SDK предоставляет полезный вспомогательный код, который находится а папке \Samples\C++\Common корневого каталога DXSDK. Среди этого кода есть класс **CD3DFont**, который отображает текст используя текстурированные треугольники и Direct3D. Поскольку **CD3DFont** использует для визуализации Direct3D, а не GDI, он работает гораздо быстрее, чем **ID3DXFont**. Однако, в отличие от **ID3DXFont**, **CD3DFont** не поддерживает сложные шрифты и форматирование. Если вам нужна скорость и достаточно простых шрифтов, класс **CD3DFont** — это ваш выбор.

Чтобы использовать класс **CD3DFont** необходимо добавить к приложению файлы d3dfont.h, d3dfont.cpp, d3dutil.h, d3dutil.cpp, dxutil.h и dxutil.cpp. Эти файлы находятся в папках Include и Src, которые расположены в ранее упоминавшейся папке Common.

9.2.1 Создание экземпляра CD3DFont

Экземпляр **CD3DFont** создается также как обычный объект C++ с помощью следующего конструктора:

```
CD3DFont(
     const TCHAR* strFontName,
     DWORD dwHeight,
     DWORD dwFlags=0L
);
```

- **strFontName** — Завершающаяся нулем строка, задающая имя гарнитуры шрифта.
- **dwHeight** — Высота шрифта.
- **dwFlags** — Необязательные дополнительные флаги; параметру можно присвоить 0 или использовать произвольную комбинацию флагов **D3DFONT_BOLD**, **D3DFONT_ITALIC**, **D3DFONT_ZENABLE**.

После создания объекта **CD3DFont** для инициализации шрифта мы должны вызвать следующие методы (в указанном порядке):

```
Font = new CD3DFont("Times New Roman", 16, 0); // создание экземпляра
Font->InitDeviceObjects(Device);
Font->RestoreDeviceObjects();
```

9.2.2 Рисование текста

Теперь, когда мы создали и инициализировали объект **CD3DFont**, можно нарисовать какой-нибудь текст. Рисование текста выполняет следующий метод:

```
HRESULT CD3DFont::DrawText(
     FLOAT x,
     FLOAT y,
     DWORD dwColor,
     const TCHAR* strText,
     DWORD dwFlags=0L
);
```

- **x** — координата x в экранном пространстве с которой начинается рисование текста.
- **y** — координата y в экранном пространстве с которой начинается рисование текста.
- **dwColor** — Цвет текста.
- **strText** — Указатель на рисуемую строку.
- **dwFlags** — Необязательные флаги визуализации; можете присвоить этому параметру 0 или указать произвольную комбинацию флагов **D3DFONT_CENTERED**, **D3DFONT_TWOSIDED**, **D3DFONT_FILTERED**.

Пример использования метода:

```
Font->DrawText(20, 20, 0xff000000, "Hello, World");
```

9.2.3 Очистка

Перед удалением объекта **CD3DFont** необходимо вызвать ряд процедур очистки, как показано в приведенном ниже фрагменте кода:

```
Font->InvalidateDeviceObjects();
Font->DeleteDeviceObjects();
delete Font;
```

9.3 D3DXCreateText

Функция **D3DXCreateText** создает трехмерную сетку, представляющую строку текста. На рис. 9.1 показана такая трехмерная сетка, отображаемая приложением FontMesh3D, которое находится в сопроводительных файлах к данной главе.

Рис. 9.1. *Трехмерный текст, созданный функцией D3DXCreateText*

Прототип функции выглядит следующим образом:

```
HRESULT D3DXCreateText(
     LPDIRECT3DDEVICE9 pDevice,
     HDC hDC,
     LPCTSTR pText,
     FLOAT Deviation,
     FLOAT Extrusion,
     LPD3DXMESH* ppMesh,
     LPD3DXBUFFER* ppAdjacency,
     LPGLYPHMETRICSFLOAT pGlyphMetrics
);
```

В случае успешного завершения функция возвращает **D3D_OK**.

- **pDevice** — Устройство, связанное с сеткой.
- **hDC** — Дескриптор контекста устройства, содержащего описание шрифта, которое будет использоваться для генерации сетки.
- **pText** — Указатель на завершающуюся нулем строку с текстом, для которого будет создаваться сетка.
- **Deviation** — Максимальное хордальное отклонение от контуров шрифта TrueType. Значение должно быть больше или равно нулю. Когда

значение равно нулю, хордальное отклонение будет равно одной проектной единице оригинального шрифта.

- **Extrusion** — Глубина шрифта, измеряемая вдоль отрицательного направления оси Z.
- **ppMesh** — Возвращает созданную сетку.
- **ppAdjacency** — Возвращает информацию о смежности для созданной сетки. Если она вам не нужна, укажите в данном параметре **null**.
- **pGlyphMetrics** — Указатель на массив структур **LPGLYPHMETRICSFLOAT**, содержащий данные метрик глифов. Каждый элемент массива содержит информацию о местоположении и ориентации соответствующего глифа в строке. Количество элементов массива должно соответствовать количеству символов в строке. Если вы не хотите связываться с метриками глифов, просто укажите 0.

Следующий фрагмент кода показывает как создать изображающую текст трехмерную сетку с помощью рассматриваемой функции.

```
// Получение дескриптора контекста устройства
HDC hdc = CreateCompatibleDC(0);

// Заполнение структуры LOGFONT, описывающей свойства шрифта
LOGFONT lf;
ZeroMemory(&lf, sizeof(LOGFONT));

lf.lfHeight    = 25;    // в логических единицах
lf.lfWidth     = 12;    // в логических единицах
lf.lfWeight    = 500;   // насыщенность,
                        // диапазон 0(тонкий) - 1000(жирный)
lf.lfItalic    = false;
lf.lfUnderline = false;
lf.lfStrikeOut = false;
lf.lfCharSet   = DEFAULT_CHARSET;
strcpy(lf.lfFaceName, "Times New Roman"); // гарнитура шрифта

// Создаем шрифт и выбираем его в контексте устройства
HFONT hFont;
HFONT hFontOld;
hFont = CreateFontIndirect(&lf);
hFontOld = (HFONT)SelectObject(hdc, hFont);

// Создаем представляющую текст трехмерную сетку
ID3DXMesh* Text = 0;
D3DXCreateText(_device, hdc, "Direct3D", 0.001f, 0.4f, &Text, 0, 0);

// Восстанавливаем бывший до этого шрифт и освобождаем ресурсы
SelectObject(hdc, hFontOld);
DeleteObject(hFont);
DeleteDC(hdc);
```

Теперь вы можете визуализировать трехмерную сетку просто вызвав метод сетки **DrawSubset**:

```
Text->DrawSubset(0);
```

9.4 Итоги

- Если вам необходима поддержка сложных шрифтов и форматирования, используйте для визуализации текста интерфейс **`ID3DXFont`**. Он использует при визуализации текста GDI и поэтому работает медленно.
- Для быстрой визуализации простого текста используйте класс **`CD3DFont`**. Он использует для визуализации текста текстурированные треугольники и Direct3D и поэтому работает гораздо быстрее, чем **`ID3DXFont`**.
- Чтобы создать трехмерную сетку, изображающую строку текста, воспользуйтесь функцией **`D3DXCreateText`**.

Глава 10
Сетки: часть I

Мы уже работали с интерфейсом **ID3DXMesh**, когда использовали функции **D3DXCreate***; в данной главе мы исследуем этот интерфейс более подробно. Почти вся глава посвящена обзору членов данных и методов, относящихся к интерфейсу **ID3DXMesh**.

Обратите внимание, что интерфейс **ID3DXMesh** наследует большую часть своей функциональности от родителя, **ID3DXBaseMesh**. Это важно знать, потому что другие интерфейсы сеток, например, **ID3DXPMesh** (прогрессивные сетки), также являются наследниками **ID3DXBaseMesh**. Следовательно, обсуждаемый в данной главе материал применим и для работы с другими типами сеток.

Цели

- Изучить внутреннюю организацию данных в объекте **ID3DXMesh**.
- Узнать, как создать **ID3DXMesh**.
- Узнать, как оптимизировать **ID3DXMesh**.
- Узнать, как визуализировать **ID3DXMesh**.

10.1 Геометрия сетки

В интерфейсе **ID3DXBaseMesh** есть буфер вершин, хранящий данные обо всех вершинах сетки, и буфер индексов, описывающий то, как вершины сетки группируются в треугольные ячейки. Мы можем получить указатели на эти буферы с помощью следующих методов:

```
HRESULT ID3DXMesh::GetVertexBuffer(LPDIRECT3DVERTEXBUFFER9* ppVB);
HRESULT ID3DXMesh::GetIndexBuffer(LPDIRECT3DINDEXBUFFER9* ppIB);
```

А вот пример того, как эти методы используются в программе:

```
IDirect3DVertexBuffer9* vb = 0;
Mesh->GetVertexBuffer(&vb);

IDirect3DIndexBuffer9* ib = 0;
Mesh->GetIndexBuffer(&ib);
```

ПРИМЕЧАНИЕ Учтите, что в качестве примитивов в интерфейсе **ID3DXMesh** поддерживаются только индексированные списки треугольников.

Также, если мы хотим заблокировать буфер для чтения или записи, можно воспользоваться приведенной ниже парой методов. Обратите внимание, что эти методы блокируют весь буфер вершин или индексов.

```
HRESULT ID3DXMesh::LockVertexBuffer(DWORD Flags, BYTE** ppData);
HRESULT ID3DXMesh::LockIndexBuffer(DWORD Flags, BYTE** ppData);
```

Параметр **Flags** указывает, как именно должна осуществляться блокировка. Флаги блокировки буферов вершин и индексов были описаны в главе 3, где мы впервые познакомились с этими буферами. Аргумент **ppData** — это адрес указателя, который после завершения работы функции будет указывать на занятую буфером область памяти.

Помните, что после завершения работы с буфером необходимо вызвать соответствующий метод для разблокировки буфера:

```
HRESULT ID3DXMesh::UnlockVertexBuffer();
HRESULT ID3DXMesh::UnlockIndexBuffer();
```

Ниже приведен список дополнительных методов интерфейса **ID3DXMesh**, которые можно использовать для получения информации о геометрии сетки:

- **DWORD GetFVF()** — Возвращает значение типа **DWORD**, описывающее формат вершин сетки.
- **DWORD GetNumVertices()** — Возвращает количество вершин в буфере вершин.
- **DWORD GetNumBytesPerVertex()** — Возвращает количество байт, занимаемых описанием одной вершины.

- **`DWORD GetNumFaces()`** — Возвращает количество граней (треугольных ячеек) в сетке.

10.2 Подгруппы и буфер атрибутов

Сетка состоит из одной или нескольких подгрупп. *Подгруппой* (*subset*) называется группа треугольных граней сетки, которая визуализируется вся с использованием одних и тех же атрибутов. Под *атрибутами* мы понимаем материал, текстуру и режим визуализации. На рис. 10.1 показано, как как можно разделить на несколько подгрупп сетку, представляющую дом.

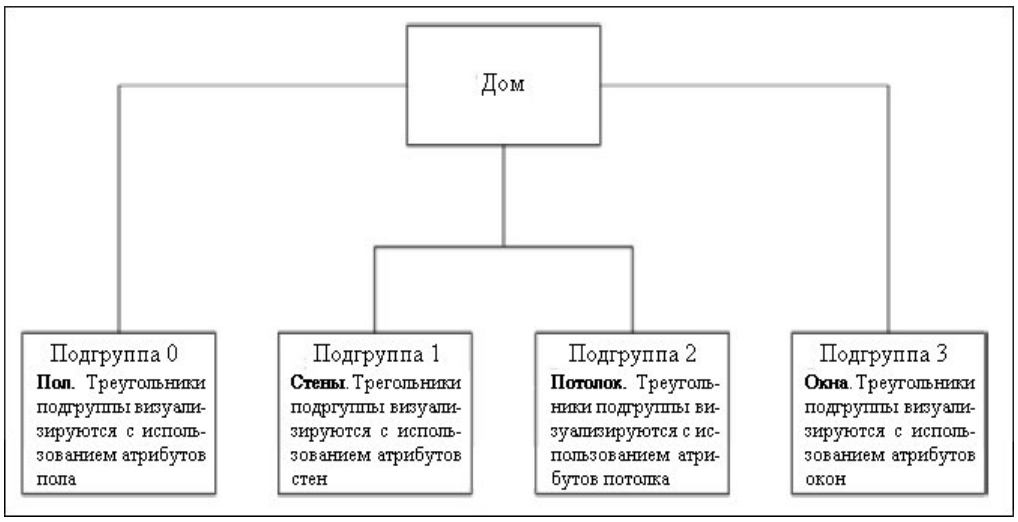

Рис. 10.1. *Дом, разбитый на несколько подгрупп*

Мы отмечаем подгруппы присваивая каждой из них уникальный положительный номер. Можно использовать любые значения, которые могут храниться в переменной типа **`DWORD`**. Например, на рис. 10.1 мы присваиваем подгруппам номера 0, 1, 2 и 3.

Каждому треугольнику сетки присваивается *идентификатор атрибута* (*attribute ID*), определяющий подгруппу, к которой относится данный треугольник. Например, на рис. 10.1 у треугольников, образующих пол дома идентификатор атрибута будет равен 0, поскольку они относятся к подгруппе 0. Аналогичным образом, у треугольников, образующих стены идентификатор атрибута будет равен 1, показывая, что они находятся в подгруппе 1.

Идентификаторы атрибутов треугольников хранятся в *буфере атрибутов* (*attribute buffer*) сетки, представляющем собой массив значений типа **`DWORD`**. Поскольку каждой грани соответствует элемент буфера атрибутов, число элементов буфера атрибутов равно количеству граней сетки. Между элементами буфера атрибутов и треугольниками, описанными в буфере индексов установлено строгое соответствие: i-ый элемент буфера атрибутов относится к *i*-ому

треугольнику из буфера индексов, образованному вершинами, на которые указывают следующие три индекса из буфера индексов:

$$A = i \times 3$$
$$B = i \times 3 + 1$$
$$C = i \times 3 + 2$$

Это соответствие показано на рис. 10.2:

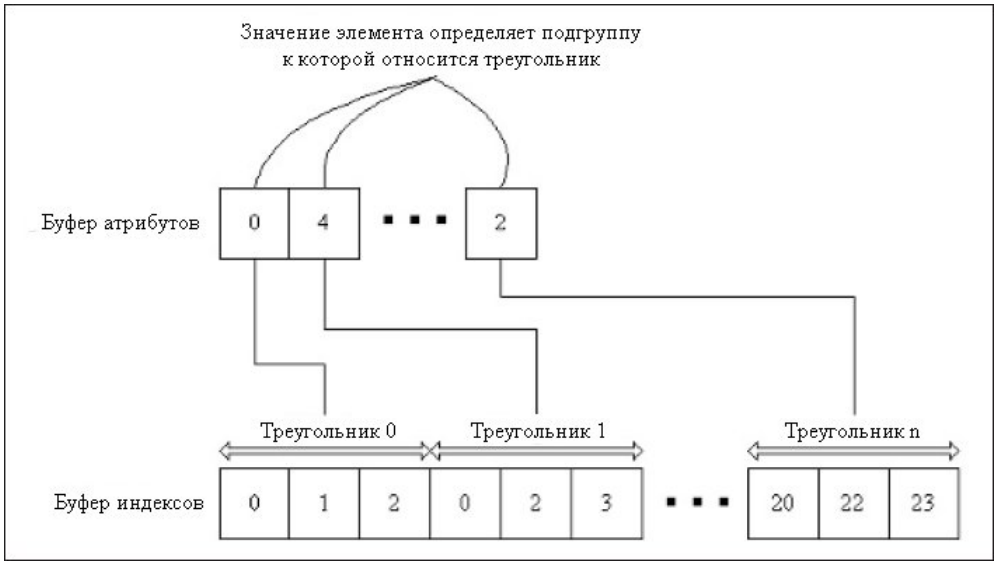

Рис. 10.2. *Соответствие между треугольниками, описываемыми элементами буфера индексов и элементами буфера атрибутов. Как видите, треугольник 0 относится к подгруппе 0, треугольник 1 — к подгруппе 4, а треугольник n — к подгруппе 2*

Мы можем получить доступ к буферу атрибутов, заблокировав его, как показано в приведенном ниже фрагменте кода:

```
DWORD* buffer = 0;
Mesh->LockAttributeBuffer(lockingFlags, &buffer);

// Чтение из буфера атрибутов или запись в него

Mesh->UnlockAttributeBuffer();
```

10.3 Рисование

Интерфейс **`ID3DXMesh`** предоставляет метод **`DrawSubset(DWORD AttribId)`**, позволяющий нарисовать все треугольники, относящиеся к подгруппе сетки, заданной аргументом **`AttribId`**.

Например, чтобы нарисовать все треугольники, относящиеся к подгруппе 0, мы должны написать:

```
Mesh->DrawSubset(0);
```

Чтобы нарисовать сетку целиком, мы должны нарисовать все входящие в нее подгруппы. Очень удобно присваивать подгруппам последовательно увеличивающиеся номера, 0, 1, 2, ..., $n - 1$, где n — это количество подгрупп, и создать соответствующие массивы материалов и текстур таким образом, чтобы i-ые элементы в массивах материалов и текстур соответствовали i-ой подгруппе. Благодаря этому можно будет визуализировать всю сетку с помощью простого цикла:

```
for(int i = 0; i < numSubsets; i++)
{
     Device->SetMaterial(mtrls[i]);
     Device->SetTexture(0, textures[i]);
     Mesh->DrawSubset(i);
}
```

10.4 Оптимизация

Можно реорганизовать вершины и индексы сетки, чтобы ее визуализация выполнялась более эффективно. Данная операция называется оптимизацией сетки, и ее можно выполнить с помощью следующего метода:

```
HRESULT ID3DXMesh::OptimizeInplace(
     DWORD Flags,
     CONST DWORD* pAdjacencyIn,
     DWORD* pAdjacencyOut,
     DWORD* pFaceRemap,
     LPD3DXBUFFER* ppVertexRemap
);
```

- **`Flags`** — Флаги оптимизации, указывающие как именно будет выполняться оптимизация. Можно использовать один или несколько из перечисленных ниже флагов:
  - **`D3DXMESHOPT_COMPACT`** — Удаляет из сетки неиспользуемые вершины и индексы.
  - **`D3DXMESHOPT_ATTRSORT`** — Сортирует треугольники по значению идентификатора атрибута и создает таблицу атрибутов. Это повышает эффективность работы метода **`DrawSubset`** (см. раздел 10.5).
  - **`D3DXMESHOPT_VERTEXCACHE`** — Увеличивает частоту попаданий кэша вершин.

- **D3DXMESHOPT_STRIPREORDER** — Реорганизует индексы таким образом, чтобы полосы треугольников были максимальной длины.
- **D3DXMESHOPT_IGNOREVERTS** — Оптимизировать только индексы, вершины игнорируются.

ПРИМЕЧАНИЕ	Флаги **D3DXMESHOPT_VERTEXCACHE** и **D3DXMESHOPT_STRIPREORDER** нельзя использовать вместе.

- **pAdjacencyIn** — Указатель на массив данных смежности граней неоптимизированной сетки.
- **pAdjacencyOut** — Указатель на массив значений типа **DWORD**, который будет заполнен информацией о смежности граней оптимизированной сетки. В массиве должно быть **ID3DXMesh::GetNumFaces() * 3** элементов. Если эта информация вам не нужна, укажите в данном параметре 0.
- **pFaceRemap** — Указатель на массив значений типа **DWORD**, который будет заполнен информацией о перемещении граней. В массиве должно быть **ID3DXMesh::GetNumFaces()** элементов. При оптимизации сетки ее грани в буфере индексов могут перемещаться. Информация о перемещении сообщает куда в результате оптимизации была перемещена данная грань оригинала; следовательно *i*-ый элемент массива **pFaceRemap** содержит индекс грани, указывающий куда была перемещена *i*-ая грань исходной неоптимизированной сетки. Если вам не нужна эта информация, укажите в данном параметре 0.
- **ppVertexRemap** — Адрес указателя на буфер **ID3DXBuffer** (см. раздел 11.1), который будет заполнен информацией о перемещении вершин. Буфер должен содержать **ID3DXMesh::GetNumVertices()** вершин. При оптимизации сетки вершины, находящиеся в буфере вершин, могут перемещаться. Информация о перемещении сообщает куда в результате оптимизации была перемещена данная вершина оригинала; следовательно *i*-ый элемент массива **ppVertexRemap** содержит индекс вершины, указывающий куда была перемещена *i*-ая вершина исходной неоптимизированной сетки. Если вам не нужна эта информация, укажите в данном параметре 0.

Пример вызова метода:

```
// Получаем информацию о смежности граней
// неоптимизированной сетки
DWORD adjacencyInfo[Mesh->GetNumFaces() * 3];
Mesh->GenerateAdjacency(0.0f, adjacencyInfo);

// Массив для хранения информации о смежности граней
// оптимизированной сетки
DWORD optimizedAdjacencyInfo[Mesh->GetNumFaces() * 3];
```

```
Mesh->OptimizeInplace(
     D3DXMESHOPT_ATTRSORT |
     D3DXMESHOPT_COMPACT |
     D3DXMESHOPT_VERTEXCACHE,
     adjacencyInfo,
     optimizedAdjacencyInfo,
     0,
     0);
```

Существует похожий метод **Optimize**, который вместо того чтобы оптимизировать исходную сетку, возвращает новый оптимизированный объект сетки и оставляет исходную сетку неизменной.

```
HRESULT ID3DXMesh::Optimize(
     DWORD Flags,
     CONST DWORD* pAdjacencyIn,
     DWORD* pAdjacencyOut,
     DWORD* pFaceRemap,
     LPD3DXBUFFER* ppVertexRemap,
     LPD3DXMESH* ppOptMesh // возвращает оптимизированную сетку
);
```

10.5 Таблица атрибутов

Когда при оптимизации сетки указан флаг **D3DXMESHOPT_ATTRSORT**, данные о геометрии сетки сортируются по значению идентификатора атрибутов. В результате данные каждой подгруппы занимают единый непрерывный участок буфера вершин и индексов (рис. 10.3).

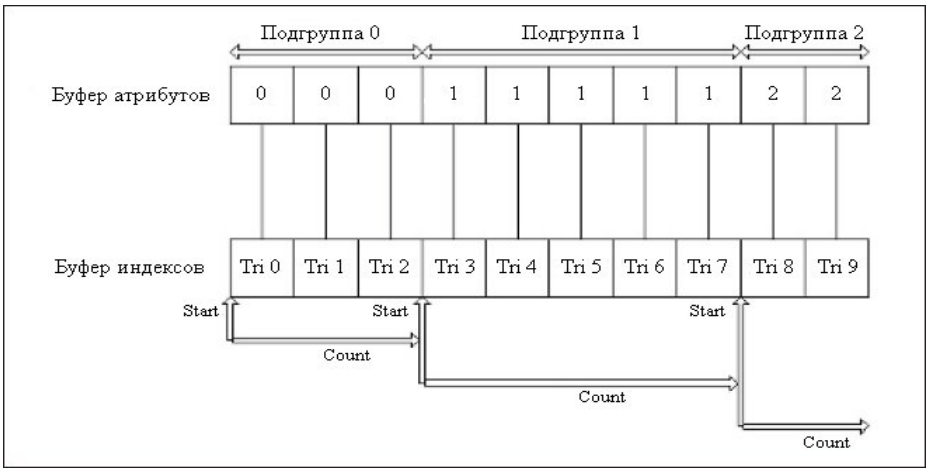

Рис. 10.3. *Данные геометрии и содержимое буфера атрибутов отсортированы по значению идентификатора атрибута, благодаря чему данные отдельной подгруппы хранятся в непрерывном фрагменте памяти. Теперь можно легко определить где начинаются и где заканчиваются данные подгруппы. Обратите внимание, что каждый блок с пометкой «Tri» в буфере индексов представлен тремя индексами*

Помимо сортировки данных о геометрии, оптимизация с флагом **D3DXMESHOPT_ATTRSORT** строит таблицу атрибутов. Таблица атрибутов представляет собой массив структур **D3DXATTRIBUTERANGE**. Каждый элемент таблицы атрибутов соответствует отдельной подгруппе сетки и определяет блок памяти в буферах индексов и вершин, где размещаются данные о геометрии данной подгруппы. Определение структуры **D3DXATTRIBUTERANGE** выглядит следующим образом:

```
typedef struct _D3DXATTRIBUTERANGE {
     DWORD AttribId;
     DWORD FaceStart;
     DWORD FaceCount;
     DWORD VertexStart;
     DWORD VertexCount;
} D3DXATTRIBUTERANGE;
```

- **AttribId** — Идентификатор подгруппы.
- **FaceStart** — Смещение в буфере индексов (**FaceStart * 3**), указывающее где в буфере расположен первый треугольник, относящийся к данной подгруппе.
- **FaceCount** — Количество граней (треугольников) в подгруппе.
- **VertexStart** — Смещение в буфере вершин, указывающее где в буфере расположена первая вершина, относящаяся к данной подгруппе.
- **VertexCount** — Количество вершин в подгруппе.

Мы можем просто просмотреть члены даных структуры **D3DXATTRIBUTERANGE**, что показано на рис. 10.3. Таблица атрибутов для сетки на рис. 10.3 состоит из трех элементов — по одному на каждую подгруппу сетки.

Если таблица атрибутов построена, визуализация подгруппы выполняется очень эффективно, достаточно быстрого просмотра таблицы атрибутов, чтобы получить всю необходимую информацию о расположении данных геометрии подгруппы. Обратите внимание, что без таблицы атрибутов визуализация сетки требует выполнения линейного поиска во всем буфере атрибутов, чтобы найти данные о геомертии рисуемой подгруппы.

Для доступа к таблице атрибутов сетки используется следующий метод:

```
HRESULT ID3DXMesh::GetAttributeTable(
     D3DXATTRIBUTERANGE* pAttribTable,
     DWORD* pAttribTableSize
);
```

Данный метод делает две веши: возвращает количество атрибутов в таблице атрибутов и заполняет массив структур **D3DXATTRIBUTERANGE** данными атрибутов.

Чтобы получить количество элементов в таблице атрибутов, мы передаем в первом аргументе 0:

```
DWORD numSubsets = 0;
Mesh->GetAttributeTable(0, &numSubsets);
```

После того, как нам стало известно количество элементов, мы можем заполнить массив значений **D3DXATTRIBUTERANGE** данными из таблицы атрибутов с помощью следующего кода:

```
D3DXATTRIBUTERANGE table = new D3DXATTRIBUTERANGE [numSubsets];
Mesh->GetAttributeTable(table, &numSubsets);
```

Мы можем явно инициализировать таблицу атрибутов с помощью метода **ID3DXMesh::SetAttributeTable**. В приведенном ниже фрагменте выполняется инициализация таблицы атрибутов для 12 подгрупп:

```
D3DXATTRIBUTERANGE attributeTable[12];

// ...заполняем массив attributeTable данными

Mesh->SetAttributeTable(attributeTable, 12);
```

10.6 Данные о смежности

Для некоторых операций с сетками, например, для оптимизации, необходимо знать какие треугольники соседствуют с данным. Эта информация хранится в *массиве данных о смежности граней* сетки (*adjacency array*).

Массив данных о смежности граней представляет собой массив значений типа **DWORD**, каждый элемент которого содержит индекс, идентифицирующий треугольную грань сетки. Например, элемент, содержащий значение i ссылается на треугольник, образованный элементами буфера индексов

$$A = i \times 3$$

$$B = i \times 3 + 1$$

$$C = i \times 3 + 2$$

Обратите внимание, что если в элементе массива хранится значение **ULONG_MAX = 4294967295**, это означает, что с данной стороной треугольника никакие грани не соседствуют. Для обозначения данного случая можно также использовать значение –1, поскольку присваивание переменной типа **DWORD** значения –1 аналогично присваиванию значения **ULONG_MAX**. Чтобы лучше понять это, вспомните, что **DWORD** — это беззнаковое 32-разрядное целое число.

Поскольку у треугольника три стороны с ним может соседствовать не более трех других треугольников (рис. 10.4).

Следовательно, в массиве данных о смежности граней должно быть **ID3DXBaseMesh::GetNumFaces() * 3** элементов — по три возможных соседа для каждой грани сетки.

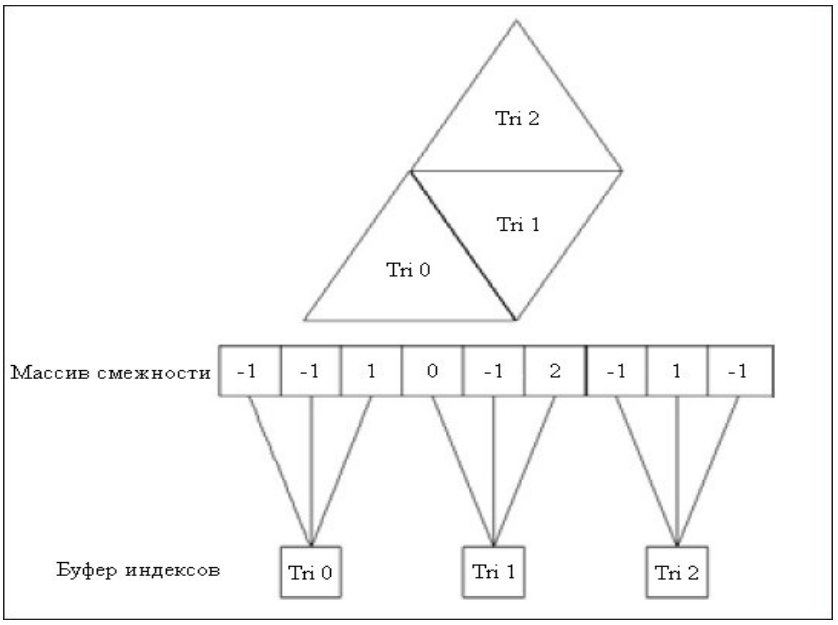

Рис. 10.4. *Каждому треугольнику соответствуют три элемента в массиве данных о смежности граней, идентифицирующие треугольники смежные с данным. Например, с треугольником Tri 1 соседствуют два треугольника (Tri 0 и Tri 2). Следовательно, в массиве данных о смежности граней треугольнику Tri 1 будут соответствовать три элемента со значениями 0, 2 и –1, указывающие что соседями данного треугольника являются Tri 0 и Tri 2. Значение –1 указывает, что у одной стороны треугольника Tri 1 нет смежных граней*

Многие из функций создания сеток библиотеки D3DX возвращают информацию о смежности граней, кроме того для ее получения можно воспользоваться следующим методом:

```
HRESULT ID3DXMesh::GenerateAdjacency(
     FLOAT fEpsilon,
     DWORD* pAdjacency
);
```

- **`fEpsilon`** — Значение, определяющее максимальное расстояние между точками, когда две различные точки будут рассматриваться как одна. То есть, если расстояние между двумя точками меньше, чем указанное значение, будет считаться, что это одна и та же точка.
- **`pAdjacency`** — Указатель на массив значений типа **`DWORD`**, который будет заполнен данными о смежности граней.

Пример использования метода:

```
DWORD adjacencyInfo[Mesh->GetNumFaces() * 3];
Mesh->GenerateAdjacency(0.001f, adjacencyInfo);
```

10.7 Клонирование

Иногда может потребоваться скопировать данные одной сетки в другую. Это можно сделать с помощью метода **ID3DXBaseMesh::CloneMeshFVF**.

```
HRESULT ID3DXMesh::CloneMeshFVF(
     DWORD Options,
     DWORD FVF,
     LPDIRECT3DDEVICE9 pDevice,
     LPD3DXMESH* ppCloneMesh
);
```

- **Options** — Один или несколько флагов, определяющих параметры создаваемого клона сетки. Полный список флагов приведен в описании перечисления **D3DXMESH** в документации SDK. Наиболее часто используются следующие флаги:
  - **D3DXMESH_32BIT** — Сетка будет использовать 32-разрядные индексы.
  - **D3DXMESH_MANAGED** — Сетка будет размещена в управляемом пуле памяти.
  - **D3DXMESH_WRITEONLY** — Данные сетки будут только записываться и не будут читаться.
  - **D3DXMESH_DYNAMIC** — Буферы сетки будут динамическими.
- **FVF** — Настраиваемый формат вершин, используемый для создаваемого клона сетки.
- **pDevice** — Устройство, связанное с клоном сетки.
- **ppCloneMesh** — Возвращает созданный клон сетки.

Обратите внимание, что метод позволяет задать для клона формат вершин отличающийся от формата вершин исходной сетки. Предположим, у нас есть сетка с форматом вершин **D3DFVF_XYZ** и мы хотим создать ее клон с форматом вершин **D3DFVF_XYZ | D3DFVF_NORMAL**. Для этого следует написать:

```
// предполагается, что _mesh и device корректные указатели
ID3DXMesh* clone = 0;
Mesh->CloneMeshFVF(
     Mesh->GetOptions(),         // используем те же параметры,
                                 // что и для исходной сетки
     D3DFVF_XYZ | D3DFVF_NORMAL, // задаем формат вершин клона
     Device,
     &clone);
```

10.8 Создание сетки (D3DXCreateMeshFVF)

До сих пор мы создавали сетки с помощью функций **D3DXCreate***. Однако мы также можем создать «пустую» сетку, воспользовавшись функцией **D3DXCreateMeshFVF**. Говоря о создании пустой сетки мы подразумеваем, что указываем количество вершин и граней, входящих в сетку, после чего функция **D3DXCreateMeshFVF** создает буферы вершин, индексов и атрибутов требуемого размера. После того, как буферы созданы, мы вручную заполняем их данными сетки (это значит, что мы должны записать данные вершин, индексы и атрибуты в буфер вершин, буфер индексов и буфер атрибутов, соответственно).

Как уже было сказано, для создания пустой сетки используется функция **D3DXCreateMeshFVF**:

```
HRESULT D3DXCreateMeshFVF(
     DWORD NumFaces,
     DWORD NumVertices,
     DWORD Options,
     DWORD FVF,
     LPDIRECT3DDEVICE9 pDevice,
     LPD3DXMESH* ppMesh
);
```

- **NumFaces** — Количество граней в создаваемой сетке. Должно быть больше нуля.
- **NumVertices** — Количество вершин в создаваемой сетке. Должно быть больше нуля.
- **Options** — Один или несколько флагов, определяющих параметры создаваемой сетки. Полный список флагов приведен в описании перечисления **D3DXMESH** в документации SDK. Наиболее часто используются следующие флаги:
  - **D3DXMESH_32BIT** — Сетка будет использовать 32-разрядные индексы.
  - **D3DXMESH_MANAGED** — Сетка будет размещена в управляемом пуле памяти.
  - **D3DXMESH_WRITEONLY** — Данные сетки будут только записываться и не будут читаться.
  - **D3DXMESH_DYNAMIC** — Буферы сетки будут динамическими.
- **FVF** — Настраиваемый формат вершин для создаваемой сетки.
- **pDevice** — Связанное с сеткой устройство.
- **ppMesh** — Возвращает созданную сетку.

В приложении, рассматриваемом в следующем разделе, будет приведен полнофункциональный пример создания сетки с помощью этой функции и заполнения данных сетки.

Вы также можете создать пустую сетку с помощью функции **D3DXCreateMesh**. Ее прототип выглядит так:

```
HRESULT D3DXCreateMesh(
     DWORD NumFaces,
     DWORD NumVertices,
     DWORD Options,
     CONST LPD3DVERTEXELEMENT9* pDeclaration,
     LPDIRECT3DDEVICE9 pDevice,
     LPD3DXMESH* ppMesh
);
```

Все параметры, за исключением четвертого, аналогичны параметрам функции **D3DXCreateMeshFVF**. Вместо того, чтобы указать формат вершин (FVF), мы передаем функции описывающий формат вершин массив структур **D3DVERTEXELEMENT9**. Сейчас мы не будем углубляться в изучение структуры **D3DVERTEXELEMENT9**; однако следует упомянуть следующую связанную функцию:

```
HRESULT D3DXDeclaratorFromFVF(
   DWORD FVF, // входной формат
   D3DVERTEXELEMENT9 Declaration[MAX_FVF_DECL_SIZE] //выходной формат
);
```

ПРИМЕЧАНИЕ Структура **D3DVERTEXELEMENT9** обсуждается в главе 17.

Эта функция получает настраиваемый формат вершин FVF и возвращает соответствующий ему массив структур **D3DVERTEXELEMENT9**. Взгляните на определение **MAX_FVF_DECL_SIZE**:

```
typedef enum {
     MAX_FVF_DECL_SIZE = 18
} MAX_FVF_DECL_SIZE;
```

10.9 Пример приложения: создание и визуализация сетки

Сопроводительный пример к данной главе визуализирует сетку куба (рис. 10.5). Приложение демонстрирует большинство действий, которые обсуждались в этой главе, включая следующие операции:

- Создание пустой сетки.
- Заполнение сетки данными о геометрии куба.
- Указание подгрупп, к которым относится каждая из граней сетки.

- Генерация информации о смежности граней сетки.
- Оптимизация сетки.
- Рисование сетки.

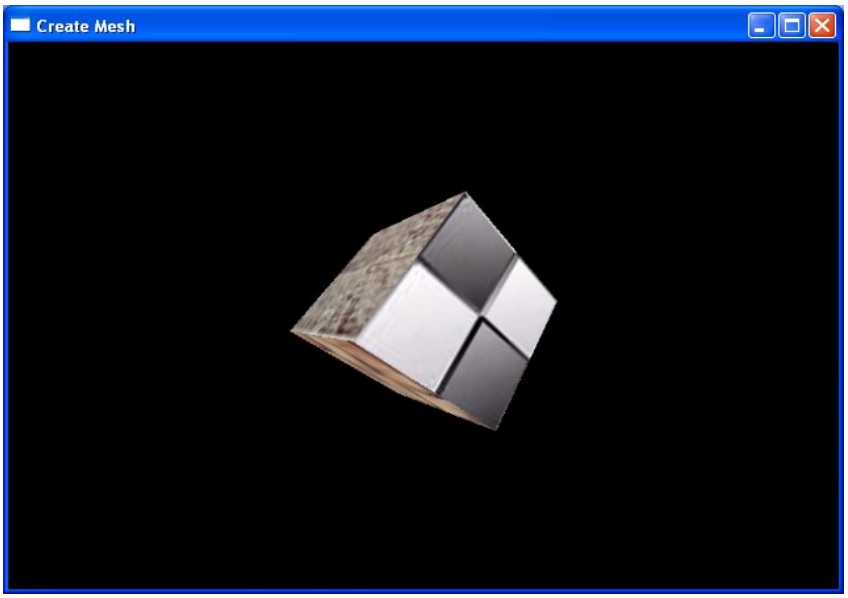

Рис. 10.5. *Куб, созданный и визуализированный как объект ID3DXMesh*

Обратите внимание, что код не относящийся к теме данной главы пропущен и не обсуждается. Полный исходный код приложения находится в сопроводительных файлах. Приложение называется D3DXCreateMeshFVF.

Кроме того, чтобы облегчить отладку и исследование компонентов сетки, мы реализуем перечисленные ниже функции, которые осуществляют вывод содержимого структур данных сетки в файл:

```
void dumpVertices(std::ofstream& outFile, ID3DXMesh* mesh);
void dumpIndices(std::ofstream& outFile, ID3DXMesh* mesh);
void dumpAttributeBuffer(std::ofstream& outFile, ID3DXMesh* mesh);
void dumpAdjacencyBuffer(std::ofstream& outFile, ID3DXMesh* mesh);
void dumpAttributeTable(std::ofstream& outFile, ID3DXMesh* mesh);
```

Имена этих функций описывают выполняемые ими действия. Поскольку реализация функций достаточно прямолинейна, мы не будем обсуждать их (посмотрите исходный код в сопроводительных файлах). В качестве примера мы рассмотрим в данном разделе функцию **`dumpAttributeTable`**.

Обзор примера мы начнем с объявлений глобальных переменных:

```
ID3DXMesh*         Mesh = 0;
const DWORD        NumSubsets = 3;
IDirect3DTexture9* Textures[3] = {0, 0, 0}; // текстуры для подгрупп
std::ofstream OutFile; // используется для вывода данных сетки в файл
```

Здесь мы объявляем указатель на сетку, которую мы создадим позже. Также мы указываем, что в создаваемой сетке будут три подгруппы. В рассматриваемом примере при визуализации каждой из подгрупп используется отдельная текстура; массив **Textures** хранит текстуры для каждой подгруппы, причем *i*-ый элемент массива текстур соответствует *i*-ой подгруппе сетки. И, наконец, переменная **OutFile** используется для вывода данных сетки в текстовый файл. Мы передаем этот объект в функции **dump***.

Основная часть работы данного приложения выполняется в функции **Setup**. Сперва мы создаем пустую сетку:

```
bool Setup()
{
     HRESULT hr = 0;
     hr = D3DXCreateMeshFVF(
           12,
           24,
           D3DXMESH_MANAGED,
           Vertex::FVF,
           Device,
           &Mesh);
```

Здесь мы выделяем память для сетки, содержащей 12 граней и 24 вершины, необходимых для описания куба.

Сейчас сетка пустая, следовательно нам необходимо записать в буфер вершин и буфер индексов данные вершин и индексы, образующие куб. Заблокируем буферы вершин и индексов и вручную запишем в них данные с помощью следующего кода:

```
// Заполнение вершин куба
Vertex* v = 0;
Mesh->LockVertexBuffer(0, (void**)&v);

// вершины передней грани куба
v[0]  = Vertex(-1.0f, -1.0f, -1.0f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f);
v[1]  = Vertex(-1.0f, 1.0f, -1.0f, 0.0f, 0.0f, -1.0f, 0.0f, 1.0f);
.
.
.
v[22] = Vertex(1.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f);
v[23] = Vertex(1.0f, -1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f);

Mesh->UnlockVertexBuffer();

// Описание треугольных граней куба
WORD* i = 0;
Mesh->LockIndexBuffer(0, (void**)&i);

// индексы граней, образующих переднюю сторону куба
i[0] = 0; i[1] = 1; i[2] = 2;
i[3] = 0; i[4] = 2; i[5] = 3;
```

```
.
.
.
// индексы граней, образующих правую сторону куба
i[30] = 20; i[31] = 21; i[32] = 22;
i[33] = 20; i[34] = 22; i[35] = 23;

Mesh->UnlockIndexBuffer();
```

Геометрия сетки определена, и мы должны не забыть указать к какой подгруппе относится каждый из образующих сетку треугольников. Вспомните, что эти сведения хранятся в буфере атрибутов. В рассматриваемом примере мы указываем, что первые четыре из описанных в буфере индексов треугольника относятся к подгруппе 0, следующие четыре треугольника — к подгруппе 1, и последние четыре треугольника (всего получается 12) — к подгруппе 2. Это делает следующий фрагмент кода:

```
DWORD* attributeBuffer = 0;
Mesh->LockAttributeBuffer(0, &attributeBuffer);

for(int a = 0; a < 4; a++)   // треугольники 1-4
     attributeBuffer[a] = 0; // подгруппа 0

for(int b = 4; b < 8; b++)   // треугольники 5-8
     attributeBuffer[b] = 1; // подгруппа 1

for(int c = 8; c < 12; c++)  // треугольники 9-12
     attributeBuffer[c] = 2; // подгруппа 2

Mesh->UnlockAttributeBuffer();
```

Теперь мы создали сетку, содержащую правильные данные. Мы уже сейчас можем визуализировать сетку, но давайте сперва ее оптимизируем. Обратите внимание, что для сетки куба оптимизация ничего не дает, но мы выполняем ее чтобы показать пример использования методов интерфейса **ID3DXMesh**. Чтобы выполнить оптимизацию сетки мы должны сначала получить данные о смежности ее граней:

```
std::vector<DWORD> adjacencyBuffer(Mesh->GetNumFaces() * 3);
Mesh->GenerateAdjacency(0.0f, &adjacencyBuffer[0]);
```

Затем можно оптимизировать сетку, как показано ниже:

```
hr = Mesh->OptimizeInplace(
      D3DXMESHOPT_ATTRSORT |
      D3DXMESHOPT_COMPACT |
      D3DXMESHOPT_VERTEXCACHE,
      &adjacencyBuffer[0],
      0, 0, 0);
```

К данному моменту инициализация сетки закончена и мы готовы визуализировать ее. Но остался еще один, последний, фрагмент кода функции **Setup**, который надо рассмотреть. В нем используются упомянутые ранее функции **dump*** для

вывода информации о сетке в текстовый файл. Предоставляемая ими возможность исследовать данные сетки помогает при отладке и при изучении внутренней структуры объекта сетки.

```
OutFile.open("MeshDump.txt");

dumpVertices(OutFile, Mesh);
dumpIndices(OutFile, Mesh);
dumpAttributeTable(OutFile, Mesh);
dumpAttributeBuffer(OutFile, Mesh);
dumpAdjacencyBuffer(OutFile, Mesh);

OutFile.close();

...Пропущены загрузка текстур, установка режимов визуализаии и т.д.

return true;
} // конец функции Setup()
```

К примеру, функция **dumpAttributeTable** записывает в файл данные из таблицы атрибутов. Вот ее реализация:

```
void dumpAttributeTable(std::ofstream& outFile, ID3DXMesh* mesh)
{
   outFile << "Attribute Table:" << std::endl;
   outFile << "----------------" << std::endl << std::endl;

   // количество элементов в таблице атрибутов
   DWORD numEntries = 0;

   mesh->GetAttributeTable(0, &numEntries);

   std::vector<D3DXATTRIBUTERANGE> table(numEntries);

   mesh->GetAttributeTable(&table[0], &numEntries);

   for(int i = 0; i < numEntries; i++)
   {
    outFile << "Entry " << i << std::endl;
    outFile << "------" << std::endl;

    outFile << "Subset ID: " << table[i].AttribId << std::endl;
    outFile << "Face Start: " << table[i].FaceStart << std::endl;
    outFile << "Face Count: " << table[i].FaceCount << std::endl;
    outFile << "Vertex Start: " << table[i].VertexStart << std::endl;
    outFile << "Vertex Count: " << table[i].VertexCount << std::endl;
    outFile << "std::endl;
   }

   outFile << std::endl << std::endl;
}
```

Ниже приведен фрагмент файла MeshDump.txt, создаваемого при работе рассматриваемого приложения, который содержит данные, записываемые функцией **dumpAttributeTable**.

```
Attribute Table:
---------------
Entry 0
------------
Subset ID: 0
Face Start: 0
Face Count: 4
Vertex Start: 0
Vertex Count: 8

Entry 1
------------
Subset ID: 1
Face Start: 4
Face Count: 4
Vertex Start: 8
Vertex Count: 8

Entry 2
------------
Subset ID: 2
Face Start: 8
Face Count: 4
Vertex Start: 16
Vertex Count: 8
```

Как видите, все соответствует тем данным, которые мы указали в коде при инициализации сетки — три подгруппы в каждую из которых входят четыре треугольника. Мы рекомендуем вам исследовать весь текст формируемого данной программой файла MeshDump.txt. Вы найдете его в папке приложения в сопроводительных файлах.

И, наконец, мы можем визуализировать сетку с помощью приведенного ниже кода; собственно говоря, мы просто перебираем в цикле все подгруппы сетки, устанавливаем для каждой из них соответствующую текстуру и затем рисуем подгруппу. Все получается так просто потому что мы задали подгруппам последовательные номера, идущие в порядке 0, 1, 2, ..., $n-1$, где n — это количество подгрупп.

```
bool Display(float timeDelta)
{
     if( Device )
     {
          // ...код обновления кадра пропущен

          Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                        0x00000000, 1.0f, 0);
          Device->BeginScene();
```

```
            for(int i = 0; i < NumSubsets; i++)
            {
                Device->SetTexture(0, Textures[i]);
                Mesh->DrawSubset(i);
            }

            Device->EndScene();
            Device->Present(0, 0, 0, 0);
        }
        return true;
}
```

10.10 Итоги

- Сетка содержит буфер вершин, буфер индексов и буфер атрибутов. Буфер вершин и буфер индексов хранят данные о геометрии сетки (данные вершин и описание образуемых ими треугольников). Буфер атрибутов содержит по одному значению для каждого треугольника, которое определяет к какой подгруппе относится данный треугольник.
- Сетка может быть оптимизирована с помощью методов **`OptimizeInplace`** или **`Optimize`**. При оптимизации выполняется реогранизация данных о геометрии сетки, чтобы повысить эффективность ее визуализации. Если оптимизация выполняется с указанием флага **`D3DXMESHOPT_ATTRSORT`**, будет создана таблица атрибутов. Таблица атрибутов позволяет визуализировать подгруппы сетки путем простого просмотра данных в элементах таблицы.
- Данные о смежности граней сетки представляют собой массив значений типа **`DWORD`**, содержащий по три значения для каждого треугольника сетки. Эти три значения соответствуют трем сторонам треугольника и указывают, какой треугольник соседствует с данной стороной.
- Можно создать пустую сетку с помощью функции **`D3DXCreateMeshFVF`**. Затем записываются данные о сетке с помощью соответствующих методов блокировки (**`LockVertexBuffer`**, **`LockIndexBuffer`** и **`LockAttributeBuffer`**).

Глава 11
Сетки: часть II

В этой главе мы продолжим изучение связанных с сетками интерфейсов, структур и функций, предоставляемых библиотекой D3DX. Основываясь на заложенном в предыдущей главе фундаменте мы перейдем к более интересным техникам, таким как загрузка и визуализация сложных трехмерных моделей, хранящихся в файле на диске, а также управление уровнем детализации сетки через интерфейс прогрессивных сеток.

Цели

- Узнать как загрузить данные из X-файла в объект **`ID3DXMesh`**.
- Изучить какую пользу приносит применение прогрессивных сеток и как использовать интерфейс прогрессивных сеток **`ID3DXPMesh`**.
- Познакомиться с ограничивающими объемами, выяснить чем они полезны и как их можно создать с помощью функций библиотеки D3DX.

11.1 ID3DXBuffer

В предыдущей главе мы упомянули интерфейс **ID3DXBuffer**, но подробно его не обсуждали. При работе с библиотекой D3DX мы будем часто сталкиваться с этим интерфейсом, так что имеет смысл познакомиться с ним поближе.

Интерфейс **ID3DXBuffer** представляет собой структуру данных общего назначения, которую библиотека D3DX использует для хранения данных в непрерывном блоке памяти. У интерфейса всего два метода:

- **LPVOID GetBufferPointer()** — Возвращает указатель на начало области с данными.
- **DWORD GetBufferSize()** — Возвращает размер буфера в байтах.

Чтобы структуру можно было применять для любых данных, используются указатели типа **void**. Это означает, что при получении хранящихся в буфере данных необходимо выполнять приведение типа. Например, функция **D3DXLoadMeshFromX** использует **ID3DXBuffer** чтобы возвратить информацию о смежности граней сетки. Поскольку данные о смежности граней хранятся в массиве значений типа **DWORD**, то, когда мы хотим использовать хранящуюся в буфере информацию о смежности граней, нам надо выполнить приведение типа буфера к массиву **DWORD**.

Вот пара примеров:

```
DWORD* info =(DWORD*)adjacencyInfo->GetBufferPointer();
D3DXMATERIAL* mtrls = (D3DXMATERIAL*)mtrlBuffer->GetBufferPointer();
```

Поскольку **ID3DXBuffer** это COM-объект, после завершения работы с ним его следует освободить, чтобы не было утечек памяти:

```
adjacencyInfo->Release();
mtrlBuffer->Release();
```

Мы можем создать пустой буфер **ID3DXBuffer** с помощью следующей функции:

```
HRESULT D3DXCreateBuffer(
     DWORD NumBytes,        // Размер буфера в байтах
     LPD3DXBUFFER *ppBuffer // Возвращает указатель на буфер
);
```

Приведенный ниже фрагмент кода создает буфер для хранения четырех целых чисел:

```
ID3DXBuffer* buffer = 0;
D3DXCreateBuffer(4 * sizeof(int), &buffer);
```

11.2 X-файлы

До сих пор мы работали с простыми геометрическими объектами, такими как сферы, цилиндры, кубы, и использовали функции **D3DXCreate***. Если вы

попытаетесь сконструировать собственный трехмерный объект, указывая координаты его вершин, то несомненно обнаружите, что это очень нудное занятие. Для того, чтобы облегчить эту скучную работу по созданию данных трехмерных объектов, были созданы специальные приложения, называемые *редакторами трехмерных моделей* (*3D modelers*). Эти редакторы позволяют создавать сложные и реалистично выглядящие сетки в визуальной интерактивной среде с помощью богатого набора инструментов, что делает процесс моделирования гораздо проще. Наиболее популярными редакторами моделей в отрасли программирования игр являются 3DS Max (*www.discreet.com*), LightWave 3D (*www.newtek.com*) и Maya (*www.aliaswavefront.com*).

Естественно, эти редакторы могут экспортировать данные созданной сетки (геометрию, материалы, анимацию и другие полезные данные) в файл. Следовательно, нам остается написать процедуру чтения файла, которая будет извлекать все данные сетки, после чего мы сможем использовать их в нашем приложении. Это очень хорошее решение, но есть и более удобный вариант. Существует формат файла с данными сетки, называемый X-файл (с расширением .X). Большинство популярных редакторов моделей могут выполнять экспорт данных в этот формат и, кроме того, существует множество программ-конвертеров, преобразующих распространенные форматы файлов сеток в файлы формата .X. Главным удобством X-файлов является то, что они являются «родным» форматом DirectX и, следовательно, библиотека D3DX без дополнительных усилий с вашей стороны поддерживает работу с X-файлами. Это значит, что библиотека D3DX предоставляет функции для чтения и записи X-файлов, а значит, если мы используем этот формат, нам не надо писать собственные процедуры чтения и записи.

ПРИМЕЧАНИЕ С сайта MSDN (*http://www.msdn.microsoft.com/*) вы можете загрузить пакет DirectX9 SDK Extra—Direct3D Tools, содержащий программы экспорта в формат .X для популярных редакторов трехмерных моделей, таких как 3DS Max, LightWave и Maya.

11.2.1 Загрузка X-файлов

Для загрузки содержащихся в X-файле данных сетки мы будем использовать показанную ниже функцию. Обратите внимание, что этот метод создает объект **`ID3DXMesh`** и загружает в него данные геометрии из X-файла.

```
HRESULT D3DXLoadMeshFromX(
     LPCSTR pFilename,
     DWORD Options,
     LPDIRECT3DDEVICE9 pDevice,
     LPD3DXBUFFER *ppAdjacency,
     LPD3DXBUFFER *ppMaterials,
     LPD3DXBUFFER* ppEffectInstances,
     PDWORD pNumMaterials,
     LPD3DXMESH *ppMesh
);
```

- **pFilename** — Имя загружаемого X-файла.
- **Options** — Один или несколько флагов, определяющих параметры создаваемой сетки. Полный список флагов приведен в описании перечисления **D3DXMESH** в документации SDK. Наиболее часто используются следующие флаги:
  - **D3DXMESH_32BIT** — Сетка будет использовать 32-разрядные индексы.
  - **D3DXMESH_MANAGED** — Сетка будет размещена в управляемом пуле памяти.
  - **D3DXMESH_WRITEONLY** — Данные сетки будут только записываться и не будут читаться.
  - **D3DXMESH_DYNAMIC** — Буферы сетки будут динамическими.
- **pDevice** — Связанное с сеткой устройство.
- **ppAdjacency** — Возвращает буфер **ID3DXBuffer**, содержащий массив значений типа **DWORD**, хранящий информацию о смежности граней сетки.
- **ppMaterials** — Возвращает буфер **ID3DXBuffer**, содержащий массив структур **D3DXMATERIAL**, хранящий данные о материалах сетки. Мы подробнее поговорим о материалах сетки в следующем разделе.
- **ppEffectInstances** — Возвращает буфер **ID3DXBuffer**, содержащий массив структур **D3DXEFFECTINSTANCE**. Мы игнорируем этот параметр и всегда будем передавать в нем 0.
- **pNumMaterials** — Возвращает количество используемых в сетке материалов (то есть количество элементов **D3DXMATERIAL** в массиве, возвращаемом через указатель **ppMaterials**).
- **ppMesh** — Возвращает созданный объект **ID3DXMesh**, заполненный данными о геометрии из X-файла.

11.2.2 Материалы в X-файле

Седьмой аргумент функции **D3DXLoadMeshFromX** возвращает количество используемых в сетке материалов, а пятый аргумент возвращает массив структур **D3DXMATERIAL**, содержащих данные этих материалов. Определение структуры **D3DXMATERIAL** выглядит так:

```
typedef struct D3DXMATERIAL {
     D3DMATERIAL9 MatD3D;
     LPSTR pTextureFilename;
} D3DXMATERIAL;
```

Это очень простая структура; она содержит базовую структуру **D3DMATERIAL9** и указатель на завершающуюся нулем строку, которая является именем файла связанной с материалом текстуры. X-файлы не содержат в себе данных текстур;

вместо этого они содержат имена файлов, которые используются для обращения к графическим файлам, содержащим реальные данные текстур. Следовательно, после загрузки X-файла с помощью функции **D3DXLoadMeshFromX** мы должны загрузить текстуры, используя указанные имена файлов. Мы покажем как это сделать в следующем разделе.

Особенно ценно, что функция **D3DXLoadMeshFromX** загружает данные из X-файла таким образом, что *i*-ый элемент в возвращаемом ею массиве **D3DXMATERIAL** соответствует *i*-ой подгруппе сетки. Соответственно подгруппы нумеруются в порядке 0, 1, 2, ..., $n-1$, где n — это количество подгрупп и материалов. Это позволяет визуализировать сетку с помощью простого цикла, перебирающего все подгруппы и визуализирующего их.

11.2.3 Пример приложения: загрузка X-файла

Теперь мы взглянем на относящийся к рассматриваемой теме код из первого примера к данной главе, который называется XFile. Этот пример загружает X-файл с именем bigship1.x, который находится в папке с DirectX SDK. Полный исходный код примера расположен в сопроводительных файлах. Окно рассматриваемой программы показано на рис. 11.1.

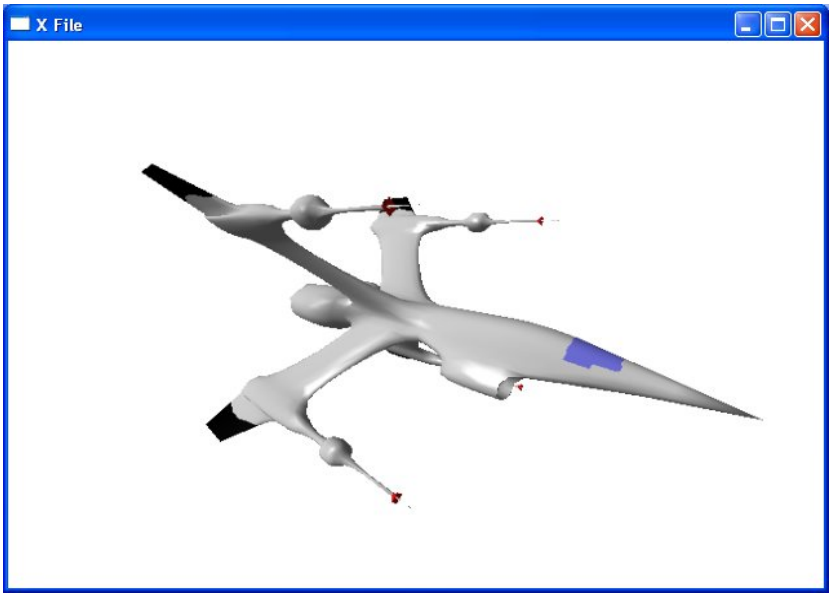

Рис. 11.1. *Окно программы XFile*

В данном примере используются следующие глобальные переменные:

```
ID3DXMesh* Mesh = 0;
std::vector<D3DMATERIAL9> Mtrls(0);
std::vector<IDirect3DTexture9*> Textures(0);
```

Здесь мы объявляем объект **ID3DXMesh**, который будет использоваться для хранения данных сетки, загружаемых из X-файла. Мы также объявляем векторы материалов и текстур, которые будут хранить используемые в сетке материалы и текстуры.

Начнем с реализации нашей стандартной функции **Setup**. Сначала мы загружаем X-файл:

```
bool Setup()
{
     HRESULT hr = 0;

     //
     // Загрузка данных из X-файла
     //
     ID3DXBuffer* adjBuffer = 0;
     ID3DXBuffer* mtrlBuffer = 0;
     DWORD numMtrls = 0;

     hr = D3DXLoadMeshFromX(
            "bigship1.x",
            D3DXMESH_MANAGED,
            Device,
            &adjBuffer,
            &mtrlBuffer,
            0,
            &numMtrls,
            &Mesh);

     if(FAILED(hr))
     {
          ::MessageBox(0, "D3DXLoadMeshFromX() - FAILED", 0, 0);
          return false;
     }
```

После того, как данные из X-файла загружены мы должны перебрать все элементы массива **D3DXMATERIAL** и загрузить текстуры, на которые ссылается сетка:

```
     //
     // Извлечение материалов и загрузка текстур
     //

     if(mtrlBuffer != 0 && numMtrls != 0)
     {
          D3DXMATERIAL* mtrls=(D3DXMATERIAL*)mtrlBuffer->
                                             GetBufferPointer();

          for(int i = 0; i < numMtrls; i++)
          {
               // При загрузке в свойстве MatD3D не устанавливается
               // значение для фонового света, поэтому установим его
               // сейчас
               mtrls[i].MatD3D.Ambient = mtrls[i].MatD3D.Diffuse;
```

```
                // Сохраняем i-ый материал
                Mtrls.push_back(mtrls[i].MatD3D);

                // Проверяем, связана ли с i-ым материалом текстура
                if( mtrls[i].pTextureFilename != 0 )
                {
                     // Да, загружаем текстуру для i-ой подгруппы
                     IDirect3DTexture9* tex = 0;
                     D3DXCreateTextureFromFile(
                          Device,
                          mtrls[i].pTextureFilename,
                          &tex);

                     // Сохраняем загруженную текстуру
                     Textures.push_back(tex);
                }
                else
                {
                     // Нет текстуры для i-ой подгруппы
                     Textures.push_back(0);
                }
           }
     }
     d3d::Release<ID3DXBuffer*>(mtrlBuffer); // закончили работу
                                             // с буфером
.
. // Пропущен код, не относящийся к теме данной главы
. // (т.е. установка освещения, матриц вида и проекции и т.д.)
.
     return true;
} // конец функции Setup()
```

В функции **`Display`** мы в каждом кадре слегка разворачиваем сетку, чтобы она вращалась. Сетка визуализируется с помощью простого цикла, поскольку ее подгруппам присвоены номера, идущие в порядке 0, 1, 2, ..., $n-1$, где n — это количество подгрупп:

```
bool Display(float timeDelta)
{
     if(Device)
     {
          //
          // Обновление: поворот сетки
          //

          static float y = 0.0f;
          D3DXMATRIX yRot;
          D3DXMatrixRotationY(&yRot, y);
          y += timeDelta;

          if( y >= 6.28f )
               y = 0.0f;
```

```
            D3DXMATRIX World = yRot;

            Device->SetTransform(D3DTS_WORLD, &World);

            //
            // Визуализация
            //

            Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                          0xffffffff, 1.0f, 0);

            Device->BeginScene();

            for(int i = 0; i < Mtrls.size(); i++)
            {
                 Device->SetMaterial(&Mtrls[i]);
                 Device->SetTexture(0, Textures[i]);
                 Mesh->DrawSubset(i);
            }

            Device->EndScene();
            Device->Present(0, 0, 0, 0);
       }
       return true;
}
```

11.2.4 Генерация нормалей вершин

Может получиться так, что в X-файле отсутствуют данные о нормалях вершин. В этом случае нам необходимо вручную вычислить нормали вершин, поскольку они необходимы для расчета освещения. Мы уже немного говорили о том, что делать в таком случае в главе 5. Однако теперь, когда мы знаем об интерфейсе **ID3DXMesh** и его родителе **ID3DXBaseMesh**, для генерации нормалей вершин произвольной сетки можно воспользоваться следующей функцией:

```
HRESULT D3DXComputeNormals(
    LPD3DXBASEMESH pMesh,   // Сетка, для которой вычисляются нормали
    const DWORD *pAdjacency // Информация о смежности граней
);
```

Эта функция генерирует нормали вершин используя усреднение нормалей. Если предоставлена информация о смежности граней, то дублирующиеся вершины будут игнорироваться. Если же информация о смежности *не* предоставлена, то дублирующиеся вершины будут получать нормали вычисленные путем усреднения нормалей тех граней, к которым они относятся. При реализации необходимо учесть, что настраиваемый формат вершин той сетки, которую мы передаем в параметре **pMesh**, должен содержать флаг **D3DFVF_NORMAL**.

Обратите внимание, что если X-файл не содержит данных нормалей вершин, в формате вершин объекта **ID3DXMesh**, создаваемого функцией **D3DXLoadMeshFromX**, флага **D3DFVF_NORMAL** не будет. Следовательно,

перед тем как вызвать функцию **D3DXComputeNormals**, мы должны клонировать сетку, указав для клона формат вершин с установленным флагом **D3DFVF_NORMAL**. Эту особенность демонстрирует приведенный ниже фрагмент кода:

```
// Флаг D3DFVF_NORMAL указан в формате вершин сетки?
if (!(pMesh->GetFVF() & D3DFVF_NORMAL))
{
     // Нет, клонируем сетку и добавляем флаг D3DFVF_NORMAL
     // к ее формату вершин:
     ID3DXMesh* pTempMesh = 0;
     pMesh->CloneMeshFVF(
                D3DXMESH_MANAGED,
                pMesh->GetFVF() | D3DFVF_NORMAL, // добавляем флаг
                Device,
                &pTempMesh);

     // Вычисляем нормали:
     D3DXComputeNormals(pTempMesh, 0);

     pMesh->Release();  // удаляем старую сетку
     pMesh = pTempMesh; // сохраняем новую сетку с нормалями
}
```

11.3 Прогрессивные сетки

Прогрессивные сетки, представленные интерфейсом **ID3DXPMesh**, позволяют упрощать сетку путем последовательности *преобразований слияния граней* (*edge collapse transformations*, ECT). Каждое такое преобразование удаляет из сетки одну вершину и одну или две грани. Поскольку преобразование является обратимым (обратное преобразование называется *расщеплением вершин* (*vertex split*)), мы можем обратить процесс упрощения в обратную сторону и восстановить первоначальное состояние сетки. Конечно же мы не можем сделать сетку более детализированной чем оригинал; можно только упростить сетку и вернуть ее в исходное состояние. На рис. 11.2 показана сетка с тремя различными *уровнями детализации* (*levels of detail*, LOD): высоким средним и низким.

В основе прогрессивных сеток лежит таже самая идея, что и у детализируемых текстур. Работая с текстурами мы отметили, что нерационально использовать текстуры с большим разрешением для мелких удаленных объектов на которых дополнительные детали все равно будут невидимы. Тоже самое верно и для сеток; небольшим, удаленным сеткам не требуется такое же большое число граней, что и крупным, близким к зрителю сеткам, потому что дополнительные детали на мелких сетках невидимы. Таким образом мы можем прекратить тратить время на визуализацию моделей с большим количеством граней в тех случаях, когда достаточно более простой модели.

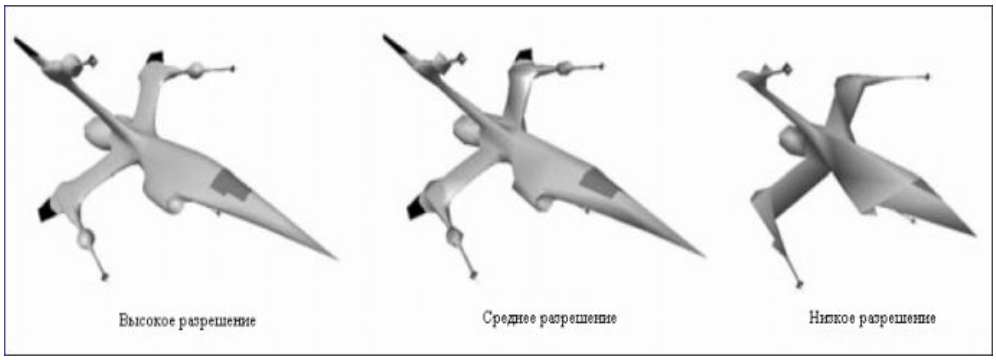

Рис. 11.2. *Изображение сетки с тремя различными уровнями детализации*

Один из способов использования прогрессивных сеток — настройка уровня детализации сетки в зависимости от расстояния между ней и камерой. При уменьшении дистанции мы добавляем детали (треугольные грани) к сетке, а когда дистанция увеличивается — мы можем убирать детали.

Обратите внимание, что мы обсуждаем не способы реализации прогрессивных сеток, а варианты использования интерфейса **ID3DXPMesh**. Тех читателей, кого интересуют детали реализации, мы отсылаем к посвященной прогрессивным сеткам статье на сайте Хьюджеса Хоппа *http://research.microsoft.com/~hoppe/*.

11.3.1 Создание прогессивной сетки

Мы можем создать объект **ID3DXPMesh** с помощью следующей функции:

```
HRESULT D3DXGeneratePMesh(
     LPD3DXMESH pMesh,
     CONST DWORD *pAdjacency,
     CONST LPD3DXATTRIBUTEWEIGHTS pVertexAttributeWeights,
     CONST FLOAT *pVertexWeights,
     DWORD MinValue,
     DWORD Options,
     LPD3DXPMESH *ppPMesh
);
```

- **pMesh** — Исходная сетка на основании данных которой будет создаваться прогрессивная сетка.
- **pAdjacency** — Указатель на массив значений типа **DWORD**, содержащий информацию о смежности граней сетки **pMesh**.
- **pVertexAttributeWeights** — Указатель на массив элементов **D3DXATTRIBUTEWEIGHTS** размера **pMesh->GetNumVertices()**, в котором *i*-ый элемент соответствует *i*-ой вершине сетки **pMesh** и задает веса ее атрибутов. *Веса атрибутов* (*attribute weight*) используются при определении того какая именно вершина будет удалена при упрощении сетки. Вы можете передать в этом параметре ноль, и тогда для каждой

вершины будут использованы веса атрибутов по умолчанию. Более подробно веса атрибутов и структура **D3DXATTRIBUTEWEIGHTS** обсуждаются в разделе 11.3.2.

- **pVertexWeights** — Указатель на массив чисел с плавающей запятой размера **pMesh->GetNumVertices()**, в котором *i*-ый элемент соответствует *i*-ой вершине сетки **pMesh** и задает вес вершины. Чем больше вес вершины, тем меньше у нее шансов, что она будет удалена в процессе упрощения сетки. Вы можете передать в этом параметре ноль и тогда вес каждой вершины будет равен 1.0 (значение по умолчанию).
- **MinValue** — Минимально возможное количество вершин или граней в сетке (что будет учитываться — вершины или грани — определяет следующий параметр **Options**) до которого может производиться упрощение. Обратите внимание, что это только желаемое значение и, в зависимости от весов вершин/атрибутов, параметры полученной в результате сетки могут не соответствовать этому значению.
- **Options** — Один из членов перечисления **D3DXMESHSIMP**:
  - **D3DXMESHSIMP_VERTEX** — Указывает, что предыдущий параметр **MinValue** задает количество вершин.
  - **D3DXMESHSIMP_FACE** — Указывает, что предыдущий параметр **MinValue** задает количество граней.
- **ppPMesh** — Возвращает созданную прогрессивную сетку.

11.3.2 Веса атрибутов вершин

```
typedef struct _D3DXATTRIBUTEWEIGHTS {
     FLOAT Position;
     FLOAT Boundary;
     FLOAT Normal;
     FLOAT Diffuse;
     FLOAT Specular;
     FLOAT Texcoord[8];
     FLOAT Tangent;
     FLOAT Binormal;
} D3DXATTRIBUTEWEIGHTS;
```

Структура данных с весами атрибутов вершины позволяет нам указать вес для каждого возможного компонента вершины. Значение 0.0 указывает, что данный компонент не обладает весом. Чем больше вес компонентов вершины, тем меньше вероятность, что данная вершина будет удалена при упрощении сетки. По умолчанию используются следующие значения:

```
D3DXATTRIBUTEWEIGHTS AttributeWeights;
AttributeWeights.Position = 1.0;
AttributeWeights.Boundary = 1.0;
AttributeWeights.Normal   = 1.0;
AttributeWeights.Diffuse  = 0.0;
AttributeWeights.Specular = 0.0;
AttributeWeights.Tex[8]   = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
```

Рекомендуется всегда использовать значения по умолчанию, только если у приложения нет веских причин отказаться от них.

11.3.3 Методы ID3DXPMesh

Интерфейс **ID3DXPMesh** является наследником интерфейса **ID3DXBaseMesh**. Поэтому он наследует всю функциональность изученного нами ранее интерфейса **ID3DXMesh**, а также предоставляет следующие дополнительные методы (учтите, что это не полный список):

- **DWORD GetMaxFaces(VOID)** — Возвращает максимальное количество граней, которое может быть в прогрессивной сетке.
- **DWORD GetMaxVertices(VOID)** — Возвращает максимальное количество вершин, которое может быть в прогрессивной сетке.
- **DWORD GetMinFaces(VOID)** — Возвращает минимальное количество граней, которое может быть в прогрессивной сетке.
- **DWORD GetMinVertices(VOID)** — Возвращает минимальное количество вершин, которое может быть в прогрессивной сетке.
- **HRESULT SetNumFaces(DWORD Faces)** — Данный метод позволяет задать количество граней до которого мы хотим упростить или усложнить сетку. Предположим, сетка состоит из 50 граней и мы хотим упростить ее до 30 граней; тогда нам следует написать:

  ```
  pmesh->SetNumFaces(30);
  ```

  Обратите внимание, что после изменения реальное количество граней сетки может отличаться от запрошенного. Если параметр **Faces** меньше, чем **GetMinFaces()**, он будет увеличен до **GetMinFaces()**. Аналогично, если **Faces** больше чем **GetMaxFaces()**, он будет уменьшен до **GetMaxFaces()**.

- **HRESULT SetNumVertices(DWORD Vertices)** — Метод позволяет задать количество вершин до которого мы хотим упростить или усложнить сетку. Предположим, сетка состоит из 20 вершин и мы хотим повысить уровень детализации, чтобы она содержала 40 вершин; тогда нам следует написать:

  ```
  pmesh->SetNumVertices(40);
  ```

  Обратите внимание, что после изменения реальное количество вершин сетки может отличаться от запрошенного. Если параметр **Vertices** меньше, чем **GetMinVertices()**, он будет увеличен до **GetMinVertices()**. Аналогично, если **Vertices** больше чем **GetMaxVertices()**, он будет уменьшен до **GetMaxVertices()**.

- ```
  HRESULT TrimByFaces(
        DWORD NewFacesMin,
        DWORD NewFacesMax,
        DWORD *rgiFaceRemap, // Перемещение граней
        DWORD *rgiVertRemap  // Перемещение вершин
  );
  ```

  Метод позволяет изменить минимальное и максимальное количество граней сетки, указав соответственно значения **NewFacesMin** и **NewFacesMax**. Обратите внимание, что новые значения должны попадать в существующий интервал от минимального до максимального значения, то есть находиться в пределах [**GetMinFaces(), GetMaxFaces()**]. Помимо этого функция возвращает информацию о перемещении вершин и граней. Данные о перемещении вершин и граней обсуждались в разделе 10.4.

- ```
  HRESULT TrimByVertices(
        DWORD NewVerticesMin,
        DWORD NewVerticesMax,
        DWORD *rgiFaceRemap, // Перемещение граней
        DWORD *rgiVertRemap  // Перемещение вершин
  );
  ```

  Метод позволяет изменить минимальное и максимальное количество вершин сетки, указав соответственно значения **NewVerticesMin** и **NewVerticesMax**. Обратите внимание, что новые значения должны попадать в существующий интервал от минимального до максимального значения, то есть находиться в пределах [**GetMinVertices(), GetMaxVertices()**]. Помимо этого функция возвращает информацию о перемещении вершин и граней. Данные о перемещении вершин и граней обсуждались в разделе 10.4.

ПРИМЕЧАНИЕ Особый интерес представляют методы **SetNumFaces** и **SetNumVertices**, поскольку они позволяют нам изменять уровень детализации сетки.

11.3.4 Пример приложения: прогрессивная сетка

Приложение Progressive Mesh очень похоже на пример XFile, за исключением того, что в нем мы создаем и визуализируем прогрессивную сетку, которая, соответственно, будет представлена интерфейсом **ID3DXPMesh**. Пользователь может изменять уровень детализации сетки с помощью клавиатуры. Чтобы увеличить количество граней сетки надо нажать клавишу **A**, а чтобы удалить грани из сетки — нажать клавишу **S**.

Используемые в данном примере глобальные переменные похожи на глобальные переменные приложения XFile, за исключением того, что мы добавили переменную для хранения прогрессивной сетки:

```
ID3DXMesh*                      SourceMesh = 0;
ID3DXPMesh*                     PMesh      = 0; // прогрессивная сетка
std::vector<D3DMATERIAL9>       Mtrls(0);
std::vector<IDirect3DTexture9*> Textures(0);
```

Вспомните, что для создания прогрессивной сетки нам надо указать исходную обычную сетку на основании данных которой будет сгенерирована прогрессивная сетка. Поэтому сперва мы загружаем данные из X-файла в объект **`ID3DXMesh`** с именем **`SourceMesh`**, и лишь потом создаем прогрессивную сетку:

```
bool Setup()
{
     HRESULT hr = 0;

     // ...Код загрузки данных из X-файла в SourceMesh пропущен
     //
     // ...Извлечение материалов и текстур пропущено
```

Поскольку данный код аналогичен коду приложения XFile мы пропустили его. Теперь, когда у нас есть исходная сетка, мы можем создать из нее прогрессивную сетку с помощью следующего кода:

```
//
// Создание прогрессивной сетки
//

hr = D3DXGeneratePMesh(
      SourceMesh,
      (DWORD*)adjBuffer->GetBufferPointer(), // смежность граней
      0,                 // использовать веса атрибутов по умолчанию
      0,                 // использовать веса вершин по умолчанию
      1,                 // упрощать насколько возможно
      D3DXMESHSIMP_FACE, // упрощать по числу граней
      &PMesh);

d3d::Release<ID3DXMesh*>(SourceMesh);  // закончили работу с сеткой
d3d::Release<ID3DXBuffer*>(adjBuffer); // закончили работу с буфером

if(FAILED(hr))
{
     ::MessageBox(0, "D3DXGeneratePMesh() - FAILED", 0, 0);
     return false;
}
```

Обратите внимание, что хотя мы и указали возможность упрощения сетки до одной грани, обычно такого упрощения не происходит из-за весов вершин и атрибутов; указание 1 приводит к упрощению сетки до минимально возможного разрешения.

Теперь прогрессивная сетка создана, но если мы ее визуализируем прямо сейчас, то она будет изображена с минимально возможным разрешением. Поскольку мы хотим визуализировать сетку с максимальным разрешением, необходимо установить его:

```
// установить максимальную детализацию
DWORD maxFaces = PMesh->GetMaxFaces();
PMesh->SetNumFaces(maxFaces);
```

В функции **Display** мы проверяем нажатие клавиш **A** и **S** и обрабатываем их:

```
bool Display(float timeDelta)
{
     if( Device )
     {
          //
          // Обновление: Смена разрешения сетки
          //

          // Получаем текущее количество граней в сетке pmesh
          int numFaces = PMesh->GetNumFaces();

          // Добавляем грань. Обратите внимание, что SetNumFaces()
          // автоматически корректирует значение, если оно
          // выходит за допустимые границы.
          if(::GetAsyncKeyState('A') & 0x8000f)
          {
              // Иногда из-за деталей внутренней реализации
              // интерфейса ID3DXPMesh, для того, чтобы инвертировать
              // преобразование слияния граней необходимо добавить
              // более одной грани. Другими словами, иногда
              // в результате добавления одной грани количество
              // граней сетки может остаться неизменным. В таком
              // случае для увеличения счетчика количества граней
              // необходимо добавить сразу две грани.
               PMesh->SetNumFaces(numFaces + 1);
               if(PMesh->GetNumFaces() == numFaces)
                    PMesh->SetNumFaces(numFaces + 2);
          }

          // Удаляем грань. Обратите внимание, что SetNumFaces()
          // автоматически корректирует значение, если оно
          // выходит за допустимые границы.
          if(::GetAsyncKeyState('S') & 0x8000f)
               PMesh->SetNumFaces(numFaces - 1);
```

Код достаточно прямолинеен, следует только обратить внимание, что иногда для инверсии преобразования слияния граней необходимо добавить к сетке не одну грань, а две.

В завершение мы визуализируем объект **ID3DXPMesh** точно так же, как визуализировали объект **ID3DXMesh**. Кроме того, мы желтыми линиями рисуем треугольные ячейки сетки, для чего рисуем сетку в каркасном режиме установив материал желтого цвета. Мы делаем это для того, чтобы можно было видеть

добавление и удаление отдельных треугольников при увеличении и уменьшении уровня детализации прогрессивной сетки.

```
            Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                          0xffffffff, 1.0f, 0);
            Device->BeginScene();

            for(int i = 0; i < Mtrls.size(); i++)
            {
                  Device->SetMaterial(&Mtrls[i]);
                  Device->SetTexture(0, Textures[i]);
                  PMesh->DrawSubset(i);

                  // рисуем каркас сетки
                  Device->SetMaterial(&d3d::YELLOW_MTRL);
                  Device->SetRenderState(D3DRS_FILLMODE,
                                         D3DFILL_WIREFRAME);
                  PMesh->DrawSubset(i);
                  Device->SetRenderState(D3DRS_FILLMODE, D3DFILL_SOLID);
            }

            Device->EndScene();
            Device->Present(0, 0, 0, 0);
      }
      return true;
} // конец функции Display
```

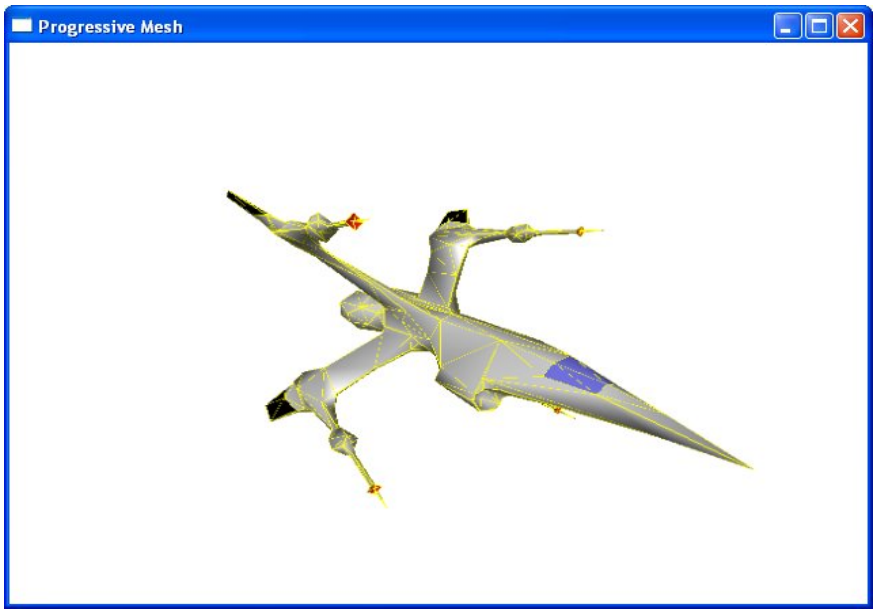

Рис. 11.3. *Окно приложения Progressive Mesh*

11.4 Ограничивающие объемы

Иногда требуется вычислить ограничивающий объем для сетки. Чаще всего в качестве ограничивающих объемов используются сферы и параллелепипеды. Иногда применяются цилиндры, элипсоиды, ромбы и капсулы. На рис. 11.4 изображена сетка с ограничивающей сферой и та же сетка с ограничивающим параллелепипедом. В данном разделе мы будем работать только с ограничивающими сферами и ограничивающими параллелепипедами.

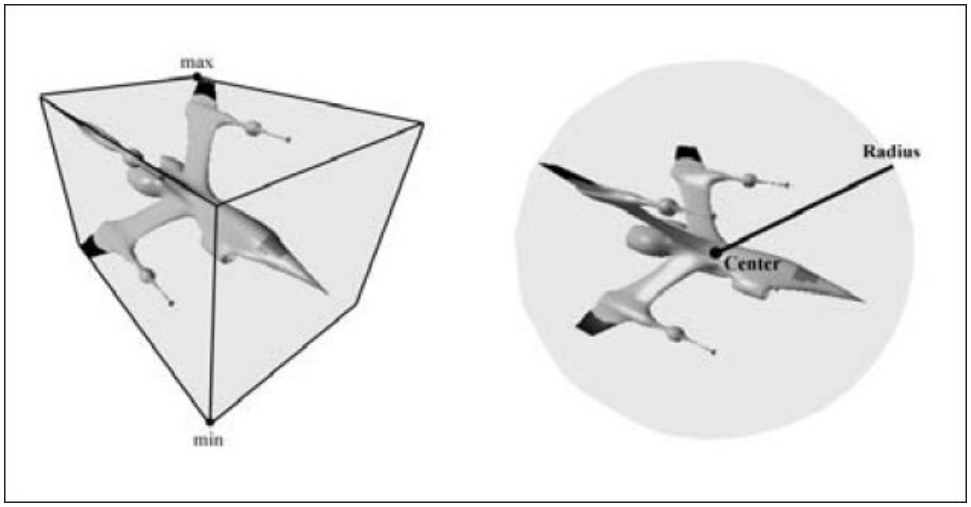

Рис. 11.4. *Сетка визуализированная с ограничивающей сферой и ограничивающим параллелепипедом. Сфера определяется путем задания центральной точки и радиуса. Параллелепипед определяется путем задания двух углов*

Ограничивающие параллелепипеды и сферы, помимо прочего, часто используются для быстрой проверки видимости объектов и для обнаружения столкновений. Например, если не видна ограничивающая сфера или ограничивающий параллелепипед сетки, значит не видна и сама сетка. Проверка видимости сферы или параллелепипеда выполняется гораздо быстрее, чем проверка видимости каждого треугольника сетки. Что касается обнаружения столкновений, предположим, что в сцене выпущена ракета и нам необходимо определить, столкнулась ли она с каким-нибудь объектом сцены. Поскольку объекты состоят из треугольных граней, нам надо перебрать каждую грань каждого объекта и проверить столкнулась ли с ней ракета (которая математически моделируется с помощью луча). Этот подход требует огромного количества проверок пересечения луча с треугольником — по одной проверке для каждой грани каждого объекта сцены. Более эффективный подход заключается в вычислении для каждой сетки ограничивающей сферы или ограничивающего параллелепипеда и последующем выполнении для каждой сетки одной проверки на пересечение луча со сферой (или параллелепипедом). Тогда мы можем сказать, что объект поражен, если луч пересекается с его ограничивающим объемом. Это достаточно хорошая аппроксимация; если требуется большая точность, мы можем

использовать проверку пересечения луча со сферой или параллелепипедом для того, чтобы отбросить те объекты, которые явно не задеты, а затем выполнить дополнительные более точные проверки для тех объектов сцены, чьи ограничивающие объемы пересекает луч.

Библиотека D3DX предоставляет функции для вычисления ограничивающей сферы и ограничивающего параллелепипеда сетки. В качестве входных данных функции получают массив вершин сетки для которой вычисляется ограничивающая сфера или ограничивающий параллелепипед. Функции достаточно гибкие и могут работать с различными форматами вершин.

```
HRESULT D3DXComputeBoundingSphere(
     LPD3DXVECTOR3 pFirstPosition,
     DWORD NumVertices,
     DWORD dwStride,
     D3DXVECTOR3* pCenter,
     FLOAT* pRadius
);
```

- **pFirstPosition** — Указатель на описывающий местоположение вектор в структуре данных первой вершины из массива вершин.
- **NumVertices** — Количество вершин в массиве вершин.
- **dwStride** — Размер данных каждой вершины в байтах. Эти сведения необходимы потому что в структуре данных вершины может храниться дополнительная информация, такая как вектор нормали или координаты текстуры, которая не требуется для вычисления ограничивающей сферы, и функция должна знать, сколько байт следует пропустить, чтобы перейти к данным местоположения следующей вершины.
- **pCenter** — Возвращает координаты центра ограничивающей сферы.
- **pRadius** — Возвращает радиус ограничивающей сферы.

```
HRESULT D3DXComputeBoundingBox(
     LPD3DXVECTOR3 pFirstPosition,
     DWORD NumVertices,
     DWORD dwStride,
     D3DXVECTOR3* pMin,
     D3DXVECTOR3* pMax
);
```

Первые три параметра те же самые, что и первые три параметра в функции **D3DXComputeBoundingSphere**. Последние два параметра используются для возврата координат двух углов ограничивающего параллелепипеда.

11.4.1 Новые константы

Давайте добавим две константы, которые будут использоваться в оставшейся части книги. Добавляются они к пространству имен **d3d**:

```
namespace d3d
{

     ...

     const float INFINITY = FLT_MAX;
     const float EPSILON  = 0.001f;
```

Константа **INFINITY** используется просто для представления наибольшего числа, которое может храниться в переменной типа **float**. Поскольку у нас не может быть значения типа **float** большего чем **FLT_MAX**, данное число будет служить для нас концепцией бесконечности, а использование именованной константы делает код более читаемым, явно указывая на те места, где подразумевается бесконечно большое значение. Константа **EPSILON** — это малое число, задав которое мы будем считать, что все числа меньше его равны нулю. Потребность в такой константе вызвана погрешностями огругления при вычислениях с плавающей точкой. Из-за них результат вычислений, который должен быть равен нулю, может немного отличаться от нуля. И, следовательно, сравнение его с нулем закончится неудачно. Поэтому для чисел с плавающей точкой мы заменяем операцию сравнения с нулем на проверку, что значение меньше, чем **EPSILON**. Приведенный ниже фрагмент кода показывает, как константа **EPSILON** используется при проверке равенства двух чисел с плавающей точкой:

```
bool Equals(float lhs, float rhs)
{
     // если lhs == rhs их разность должна быть равна нулю
     return fabs(lhs - rhs) < EPSILON ? true : false;
}
```

11.4.2 Типы ограничивающих объемов

Чтобы упростить работу с ограничивающими сферами и ограничивающими параллелепипедами мы реализуем представляющие их классы в пространстве имен **d3d**:

```
struct BoundingBox
{
     BoundingBox();

     bool isPointInside(D3DXVECTOR3& p);

     D3DXVECTOR3 _min;
     D3DXVECTOR3 _max;
};

struct BoundingSphere
{
     BoundingSphere();
```

```
     D3DXVECTOR3 _center;
     float       _radius;
};

d3d::BoundingBox::BoundingBox()
{
     // ограничивающий параллелепипед
     // бесконечно малого размера
     _min.x = d3d::INFINITY;
     _min.y = d3d::INFINITY;
     _min.z = d3d::INFINITY;

     _max.x = -d3d::INFINITY;
     _max.y = -d3d::INFINITY;
     _max.z = -d3d::INFINITY;
}

bool d3d::BoundingBox::isPointInside(D3DXVECTOR3& p)
{
     // точка внутри ограничивающего параллелепипеда?
     if(p.x >= _min.x && p.y >= _min.y && p.z >= _min.z &&
        p.x <= _max.x && p.y <= _max.y && p.z <= _max.z)
     {
          return true;
     }
     else
     {
          return false;
     }
}

d3d::BoundingSphere::BoundingSphere()
{
     _radius = 0.0f;
}
```

11.4.3 Пример приложения: ограничивающие объемы

Приложение Bounding Volumes находящееся в папке примеров к данной главе, расположенной на диске с сопроводительными файлами, демонстрирует использование функций **D3DXComputeBoundingSphere** и **D3DXComputeBoundingBox**. Программа загружает сетку из X-файла и вычисляет для нее ограничивающую сферу и ограничивающий параллелепипед. Затем программа создает два объекта **ID3DXMesh** — один для моделирования ограничивающей сферы и другой для моделирования ограничивающего параллелепипеда. После этого визуализируется загруженная из X-файла сетка и вместе с ней либо сетка ограничивающей сферы, либо сетка ограничивающего параллелепипеда (рис. 11.5). Пользователь может выбирать, что именно (ограничивающая сфера или ограничивающий параллелепипед) отображается, нажимая на клавишу **Space**.

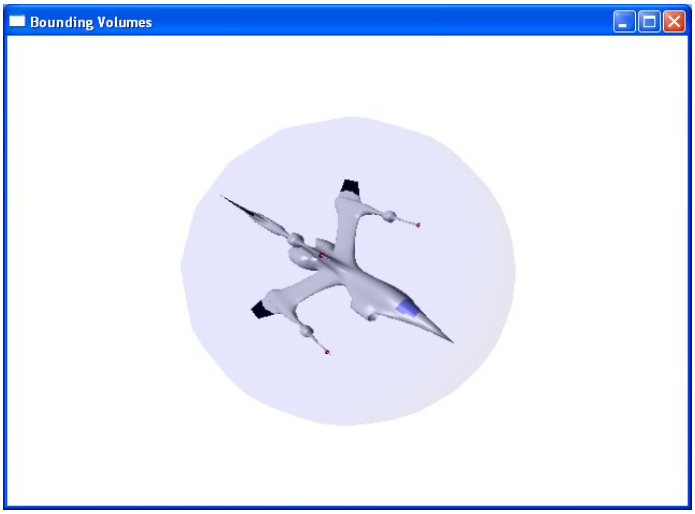

Рис. 11.5. *Окно программы Bounding Volumes. Обратите внимание, чтобы ограничивающая сфера была прозрачной используется альфа-смешивание*

Код приложения достаточно прост, и мы не будем его обсуждать. Интерес представляет только реализация двух функций, которые формируют ограничивающую сферу и ограничивающий параллелепипед для указанной сетки:

```
bool ComputeBoundingSphere(
        ID3DXMesh* mesh, // сетка, для которой вычисляется
                         // ограничивающая сфера
        d3d::BoundingSphere* sphere) // возвращает
                                     // ограничивающую сферу
{
     HRESULT hr = 0;

     BYTE* v = 0;
     mesh->LockVertexBuffer(0, (void**)&v);
     hr = D3DXComputeBoundingSphere(
            (D3DXVECTOR3*)v,
            mesh->GetNumVertices(),
            D3DXGetFVFVertexSize(mesh->GetFVF()),
            &sphere->_center,
            &sphere->_radius);
     mesh->UnlockVertexBuffer();

     if( FAILED(hr) )
          return false;

     return true;
}

bool ComputeBoundingBox(
        ID3DXMesh* mesh, // сетка, для которой вычисляется
                         // ограничивающий параллелепипед
        d3d::BoundingBox* box) // возвращает ограничивающий
                               // параллелепипед
```

```
{
     HRESULT hr = 0;

     BYTE* v = 0;
     mesh->LockVertexBuffer(0, (void**)&v);

     hr = D3DXComputeBoundingBox(
            (D3DXVECTOR3*)v,
           mesh->GetNumVertices(),
           D3DXGetFVFVertexSize(mesh->GetFVF()),
           &box->_min,
           &box->_max);

     mesh->UnlockVertexBuffer();

     if( FAILED(hr) )
          return false;

     return true;
}
```

Обратите внимание, что приведение типа **(D3DXVECTOR3*)v** подразумевает, что в используемой структуре данных вершины информация о координатах вершины хранится в самом начале. Также обратите внимание на использование функции **D3DXGetFVFVertexSize** для получения размера структуры данных вершины, соответствующей указанному описанию формата вершин.

11.5 Итоги

- Можно создать сложную сетку с помощью программ редактирования трехмерных моделей и затем либо экспортировать либо конвертировать ее в X-файл. Затем, с помощью функции **D3DXLoadMeshFromX** можно загрузить данные сетки из X-файла в объект **ID3DXMesh**, который можно использовать в приложении.
- Прогрессивные сетки, представленные интерфейсом **ID3DXPMesh**, могут применяться для управления уровнем детализации сетки; то есть мы можем динамически изменять детализацию сетки. Эта возможность очень полезна, поскольку часто требуется управление детализацией сетки в зависимости от ее местоположения в сцене. Например, расположенные близко к зрителю сетки должны отображаться с большим количеством деталей, чем сетки, находящиеся вдалеке.
- Мы можем вычислить ограничивающую сферу и ограничивающий параллелепипед с помощью функций **D3DXComputeBoundingSphere** и **D3DXComputeBoundingBox** соответственно. Ограничивающие объемы полезны, потому что помогают приблизительно оценить занимаемое сеткой пространство, что значительно ускоряет некоторые вычисления.

Глава 12
Построение гибкого класса камеры

До сих пор для вычисления матрицы преобразования пространства вида мы пользовались функцией **`D3DXMatrixLookAtLH`**. Она вполне подходит для размещения и нацеливания неподвижной камеры, но пользовательский интерфейс часто требует, чтобы камера перемещалась в ответ на действия пользователя. Поэтому нам необходимо разработать собственное решение. В данной главе мы посмотрим как реализовать класс **`Camera`**, предоставляющий большую свободу управления камерой, чем функция **`D3DXMatrixLookAtLH`**, и подходящий для авиасимуляторов и игр с видом от первого лица.

Цели

- Изучить реализацию гибкого класса **`Camera`**, который может использоваться в авиасимуляторах и играх с видом от первого лица.

12.1 Проектирование класса камеры

Мы определяем местоположение и ориентацию камеры относительно мировой системы координат с помощью четырех *векторов камеры* (*camera vectors*): *правого вектора* (*right vector*), *верхнего вектора* (*up vector*), *вектора взгляда* (*look vector*) и *вектора местоположения* (*position vector*), как показано на рис. 12.1. Эти векторы образуют локальную систему координат камеры, описанную в мировой системе координат. Поскольку правый вектор, верхний вектор и вектор взгляда описывают ориентацию камеры в мировом пространстве, мы иногда будем называть их *векторами ориентации* (*orientation vectors*). Векторы ориентации должны быть *ортонормальными*. Набор векторов называется ортонормальным, если каждый вектор перпендикулярен остальным и длина всех векторов равна единице. Причина данного ограничения в том, что позже мы подставим эти векторы в строки матрицы, а матрица в которой векторы-строки являются ортонормальными будет ортогональной. Вспомните, что отличительной особенностью ортогональной матрицы является то, что результат ее транспонирования равен результату ее инвертирования. Пользу этой особенности мы увидим в разделе 12.2.1.2.

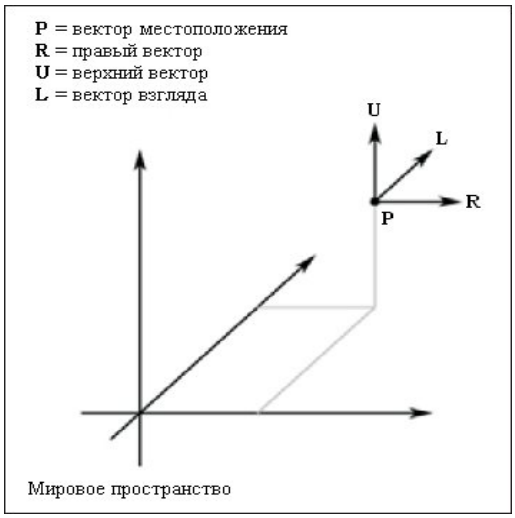

Рис. 12.1. *Векторы камеры определяют ее ориентацию и местоположение в мировой системе координат*

Эти четыре описывающих камеру вектора позволяют выполнять с камерой следующие шесть операций:

- Поворот относительно правого вектора (наклон).
- Поворот относительно верхнего вектора (отклонение).
- Поворот относительно вектора взгляда (вращение).
- Сдвиг вдоль правого вектора.
- Подъем вдоль верхнего вектора.
- Передвижение вдоль вектора взгляда.

Эти шесть операций позволяют нам перемещать камеру вдоль трех осей и вращать ее вокруг этих же осей, что дает в совокупности шесть степеней свободы. Приведенное ниже определение класса **Camera** отражает наше описание данных и требуемые методы:

```
class Camera
{
public:
     enum CameraType { LANDOBJECT, AIRCRAFT };

     Camera();
     Camera(CameraType cameraType);
     ~Camera();

     void strafe(float units); // влево/вправо
     void fly(float units);    // вверх/вниз
     void walk(float units);   // вперед/назад

     void pitch(float angle);  // вращение относительно
                               // правого вектора
     void yaw(float angle);    // вращение относительно
                               // верхнего вектора
     void roll(float angle);   // вращение относительно
                               // вектора взгляда

     void getViewMatrix(D3DXMATRIX* V);
     void setCameraType(CameraType cameraType);
     void getPosition(D3DXVECTOR3* pos);
     void setPosition(D3DXVECTOR3* pos);
     void getRight(D3DXVECTOR3* right);
     void getUp(D3DXVECTOR3* up);
     void getLook(D3DXVECTOR3* look);

private:
     CameraType  _cameraType;
     D3DXVECTOR3 _right;
     D3DXVECTOR3 _up;
     D3DXVECTOR3 _look;
     D3DXVECTOR3 _pos;
};
```

В этом определении класса есть одна вещь, которую мы до сих пор не обсуждали — перечисление **CameraType**. Дело в том, что наша камера поддерживает две модели поведения: **LANDOBJECT** и **AIRCRAFT**. Модель **AIRCRAFT** позволяет свободно перемещаться в пространстве и предоставляет шесть степеней свободы. В то же время в ряде игр с видом от первого лица персонаж не может летать, и нам надо ограничить перемещение по соответствующей оси. Чтобы внести эти ограничения, достаточно указать для камеры тип **LANDOBJECT**, что будет показано в последующих разделах.

12.2 Детали реализации

12.2.1 Вычисление матрицы вида

Сейчас мы покажем, как можно вычислить матрицу преобразования вида, на основании заданных векторов камеры. Предположим, что векторы $\mathbf{p} = (p_x, p_y, p_z)$, $\mathbf{r} = (r_x, r_y, r_z)$, $\mathbf{u} = (u_x, u_y, u_z)$ и $\mathbf{d} = (d_x, d_y, d_z)$ являются соответственно вектором местоположения, правым вектором, верхним вектором и вектором взгляда камеры.

Вспомните, что в главе 2 мы говорили о том, что преобразование пространства вида трансформирует геометрию мира таким образом, что камера помещается в начало координат и ее оси совпадают с осями мировой системы координат (рис. 12.2).

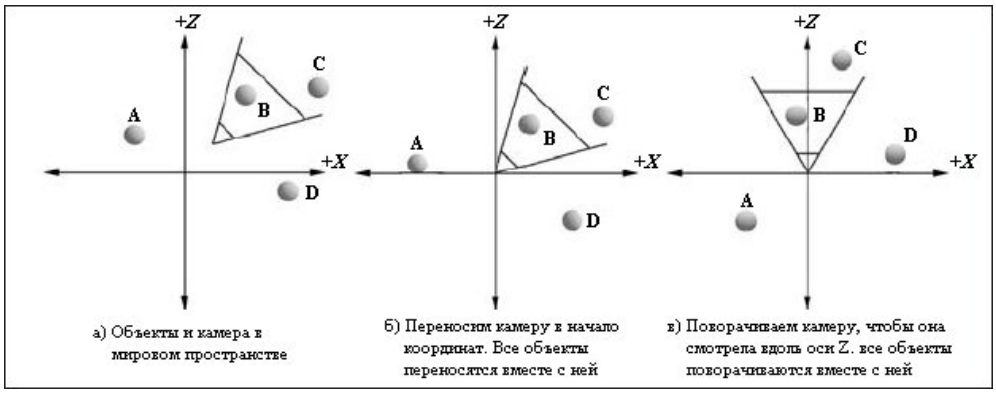

Рис. 12.2. *Преобразование из мирового пространства в пространство вида. В результате этого преобразования камера перемещается в начало координат и поворачивается так, чтобы быть направленной вдоль положительного направления оси Z. Обратите внимание, что все объекты сцены также подвергаются этому преобразованию, так что формируемый камерой вид сцены не изменяется*

Следовательно, нам нужна матрица преобразования $\mathbf{V}$, отвечающая следующим требованиям:

- $\mathbf{pV} = (0, 0, 0)$ — Матрица преобразования $\mathbf{V}$ перемещает камеру в начало координат.
- $\mathbf{rV} = (1, 0, 0)$ — Матрица $\mathbf{V}$ преобразует правый вектор камеры таким образом, чтобы он совпадал с осью X мировой системы координат.
- $\mathbf{uV} = (0, 1, 0)$ — Матрица $\mathbf{V}$ преобразует верхний вектор камеры таким образом, чтобы он совпадал с осью Y мировой системы координат.
- $\mathbf{dV} = (0, 0, 1)$ — Матрица $\mathbf{V}$ преобразует вектор взгляда камеры таким образом, чтобы он совпадал с осью Z мировой системы координат.

Задачу нахождения такой матрицы можно разделить на две части: 1) перемещение, в результате которого камера окажется в начале системы координат и 2) поворот, в результате которого оси камеры будут совпадать с осями мировой системы координат.

12.2.1.1. Часть 1: Перемещение

Чтобы точка **p** совпадала с началом координат, ее необходимо переместить на –**p**, поскольку $\mathbf{p} - \mathbf{p} = 0$. Так что отвечающая за перемещение часть матрицы преобразования вида задается следующей матрицей:

$$\mathbf{T} = \begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ -p_x & -p_y & -p_z & \mathbf{1} \end{bmatrix}$$

12.2.1.2. Часть 2: Вращение

Выравнивание трех векторов камеры по осям мировой системы координат требует большего объема работ. Нам необходима мтрица поворота **A**, размером 3 × 3, которая выравнивает правый вектор, верхний вектор и вектор взгляда камеры с осями X, Y и Z мировой системы координат. Такая матрица должна удовлетворять следующим трем выражениям:

$$\mathbf{rA} = \begin{bmatrix} r_x, & r_y, & r_z \end{bmatrix} \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{1}, & \mathbf{0}, & \mathbf{0} \end{bmatrix}$$

$$\mathbf{uA} = \begin{bmatrix} u_x, & u_y, & u_z \end{bmatrix} \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{0}, & \mathbf{1}, & \mathbf{0} \end{bmatrix}$$

$$\mathbf{dA} = \begin{bmatrix} d_x, & d_y, & d_z \end{bmatrix} \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{0}, & \mathbf{0}, & \mathbf{1} \end{bmatrix}$$

ПРИМЕЧАНИЕ Мы работаем с матрицами размера 3 × 3 потому что для представления вращения нам не нужны однородные координаты. Позднее мы вернемся к привычным матрицам размера 4 × 4.

Поскольку во всех трех приведенных выше формулах коэффициенты матрицы **A** одни и те же, можно свести все три формулы в одну, переписав их следующим образом:

$$\mathbf{BA} = \begin{bmatrix} r_x & r_y & r_z \\ u_x & u_y & u_z \\ d_x & d_y & d_z \end{bmatrix} \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix}$$

Существует несколько способов вычисления матрицы **A**, но легко заметить, что матрица **A** является инверсией матрицы **B** потому что $\mathbf{BA} = \mathbf{BB}^{-1} = \mathbf{I}$. Поскольку матрица **B** является ортогональной (ее векторы-строки являются ортонормальными), ее инверсия совпадает с результатом транспонирования. Следовательно, преобразование, выравнивающее векторы ориентации по осям мировой системы координат, выглядит так:

$$\mathbf{B}^{-1} = \mathbf{B}^{T} = \mathbf{A} = \begin{bmatrix} r_x & u_x & d_x \\ r_y & u_y & d_y \\ r_z & u_y & u_z \end{bmatrix}$$

12.2.1.3. Комбинирование обеих частей

Теперь мы дополняем **A** до матрицы 4 × 4 и комбинируем матрицу перемещения с матрицей вращения, получая матрицу преобразования вида **V**:

$$\mathbf{TA} = \begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ -p_x & -p_y & -p_z & \mathbf{1} \end{bmatrix} \begin{bmatrix} r_x & u_x & d_x & \mathbf{0} \\ r_y & u_y & d_y & \mathbf{0} \\ r_z & u_z & d_z & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix} = \begin{bmatrix} r_x & u_x & d_x & \mathbf{0} \\ r_y & u_y & d_y & \mathbf{0} \\ r_z & u_z & d_z & \mathbf{0} \\ -\mathbf{p}\cdot\mathbf{r} & -\mathbf{p}\cdot\mathbf{u} & -\mathbf{p}\cdot\mathbf{d} & \mathbf{1} \end{bmatrix} = \mathbf{V}$$

Мы вычисляем эту матрицу с помощью метода **`Camera::getViewMatrix`**:

```
void Camera::getViewMatrix(D3DXMATRIX* V)
{
     // Делаем оси камеры ортогональными
     D3DXVec3Normalize(&_look, &_look);

     D3DXVec3Cross(&_up, &_look, &_right);
     D3DXVec3Normalize(&_up, &_up);

     D3DXVec3Cross(&_right, &_up, &_look);
     D3DXVec3Normalize(&_right, &_right);

     // Строим матрицу вида:
     float x = -D3DXVec3Dot(&_right, &_pos);
     float y = -D3DXVec3Dot(&_up, &_pos);
     float z = -D3DXVec3Dot(&_look, &_pos);

     (*V)(0, 0) = _right.x;
     (*V)(0, 1) = _up.x;
     (*V)(0, 2) = _look.x;
     (*V)(0, 3) = 0.0f;

     (*V)(1, 0) = _right.y;
     (*V)(1, 1) = _up.y;
     (*V)(1, 2) = _look.y;
     (*V)(1, 3) = 0.0f;
```

```
      (*V)(2, 0) = _right.z;
      (*V)(2, 1) = _up.z;
      (*V)(2, 2) = _look.z;
      (*V)(2, 3) = 0.0f;

      (*V)(3, 0) = x;
      (*V)(3, 1) = y;
      (*V)(3, 2) = z;
      (*V)(3, 3) = 1.0f;
}
```

Возможно, вы не понимаете для чего предназначены первые строки кода данной функции. После нескольких поворотов из-за погрешности округления в операциях с плавающей точкой оси камеры могут стать неортогональными. Поэтому каждый раз при вызове данной функции мы заново вычисляем верхний и правый вектор на основании вектора взгляда, чтобы гарантировать, что все три вектора будут ортогональными. Новый ортогональный верхний вектор вычисляется по формуле *up* = *look* × *right*. Затем вычисляется новый ортогональный правый вектор по формуле *right* = *up* × *look*.

12.2.2 Вращение относительно произвольной оси

Чтобы реализовать методы для поворота нашей камеры, нам необходима возможность вращать ее относительно произвольной оси. Для этой цели библиотека D3DX предоставляет следующую функцию:

```
D3DXMATRIX *D3DXMatrixRotationAxis(
     D3DXMATRIX *pOut,      // возвращает матрицу вращения
     CONST D3DXVECTOR3 *pV, // ось вращения
     FLOAT Angle            // угол поворота в радианах
);
```

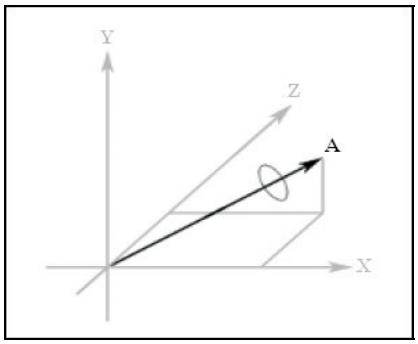

Рис. 12.3. *Вращение относительно произвольной оси, определенной вектором* ***A***

Предположим, мы хотим выполнить поворот на π/2 радиан вокруг оси, заданной вектором (0.707, 0.707, 0). Для этого надо написать:

```
D3DXMATRIX R;
D3DXVECTOR3 axis(0.707f, 0.707f, 0.0f);
D3DXMatrixRotationAxis(&R, &axis, D3DX_PI / 2.0f);
```

Формулу, по которой функция **D3DXMatrixRotationAxis** вычисляет матрицу вращения вы найдете в книге Эрика Ленджела «Mathematics for 3D Game Programming & Computer Graphics».

12.2.3 Наклон, рыскание и вращение

Поскольку векторы ориентации описывают ориентацию камеры относительно мировой системы координат, мы должны вычислять как они изменяются при наклоне, рыскании и вращении камеры. Сделать это очень просто. Взгляните на рис. 12.4, 12.5 и 12.6, где изображен наклон, рыскание и вращение камеры соответственно.

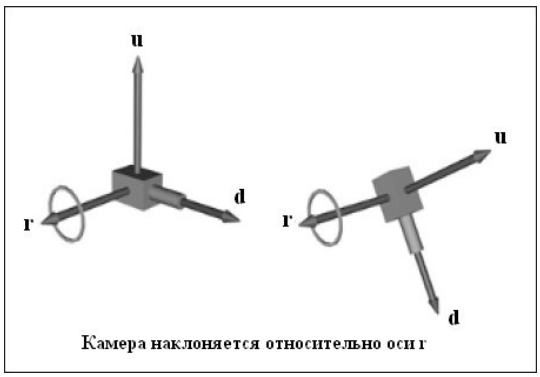

Рис. 12.4. *Наклон, или поворот относительно правого вектора камеры*

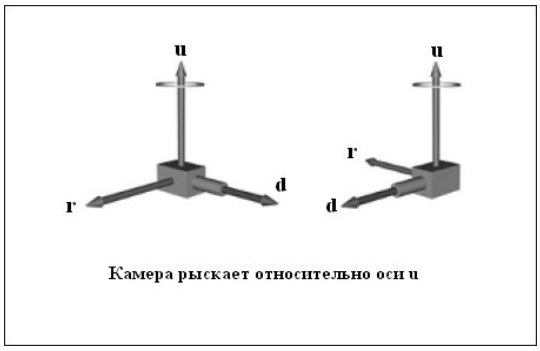

Рис. 12.5. *Рыскание, или поворот относительно верхнего вектора камеры*

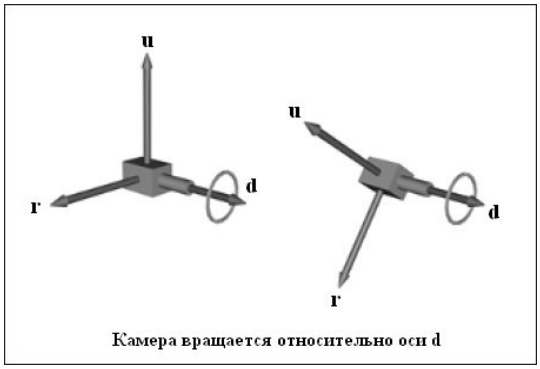

Рис. 12.6. *Вращение, или поворот относительно вектора взгляда камеры*

Как видите при наклоне камеры мы должны повернуть верхний вектор и вектор взгляда относительно правого вектора на заданный угол. Аналогично при рыскании мы поворачиваем правый вектор и вектор взгляда относительно верхнего вектора на указанный угол. А при вращении камеры мы поворачиваем на заданный угол верхний и правый векторы относительно вектора взгляда.

Теперь вы видите, почему нам необходима функция **D3DXMatrixRotationAxis**, ведь любой из трех векторов относительно которых выполняется поворот, в мировой системе координат может иметь произвольную ориентацию.

Реализация методов для наклона, рыскания и вращения соответствует приведенному выше описанию. Однако для типа камеры **LANDOBJECT** добавлен ряд ограничений. В частности, неестественно выглядит отклонение от заданного курса при наклоне или вращении наземного объекта. Поэтому при рыскании в модели **LANDOBJECT** мы выполняем поворот относительно оси Y мировой системы координат, а не относительно верхнего вектора камеры. Кроме того, мы полностью запрещаем вращение наземных объектов. Помните об этом, когда будете использовать класс **Camera** в своих собственных приложениях; мы предлагаем вам только пример.

Код реализующий наклон, рыскание и вращение камеры выглядит так:

```
void Camera::pitch(float angle)
{
     D3DXMATRIX T;
     D3DXMatrixRotationAxis(&T, &_right, angle);

     // Поворот векторов _up и _look относительно вектора _right
     D3DXVec3TransformCoord(&_up,&_up, &T);
     D3DXVec3TransformCoord(&_look,&_look, &T);
}

void Camera::yaw(float angle)
{
     D3DXMATRIX T;

     // Для наземных объектов выполняем вращение
     // вокруг мировой оси Y (0, 1, 0)
     if(_cameraType == LANDOBJECT)
          D3DXMatrixRotationY(&T, angle);

     // Для летающих объектов выполняем вращение
     // относительно верхнего вектора
     if(_cameraType == AIRCRAFT)
          D3DXMatrixRotationAxis(&T, &_up, angle);

     // Поворот векторов _right и _look относительно
     // вектора _up или оси Y
     D3DXVec3TransformCoord(&_right, &_right, &T);
     D3DXVec3TransformCoord(&_look, &_look, &T);
}
```

```
void Camera::roll(float angle)
{
     // Вращение только для летающих объектов
     if(_cameraType == AIRCRAFT)
     {
          D3DXMATRIX T;
          D3DXMatrixRotationAxis(&T, &_look, angle);
          // Поворот векторов _up и _right относительно
          // вектора _look
          D3DXVec3TransformCoord(&_right, &_right, &T);
          D3DXVec3TransformCoord(&_up, &_up, &T);
     }
}
```

12.2.4 Ходьба, сдвиг и полет

Говоря о ходьбе мы подразумеваем перемещение в направлении взгляда (то есть вдоль вектора взгляда). Сдвиг — это перемещение в сторону относительно направления взгляда, то есть перемещение вдоль правого вектора. Ну и когда мы говорим о полете, подраумевается перемещение вдоль верхнего вектора. Чтобы переместиться вдоль одной из этих осей мы просто прибавляем к вектору местоположения камеры вектор заданной длины, указывающий в том же направлении, что и ось, вдоль которой перемещается камера (рис. 12.7).

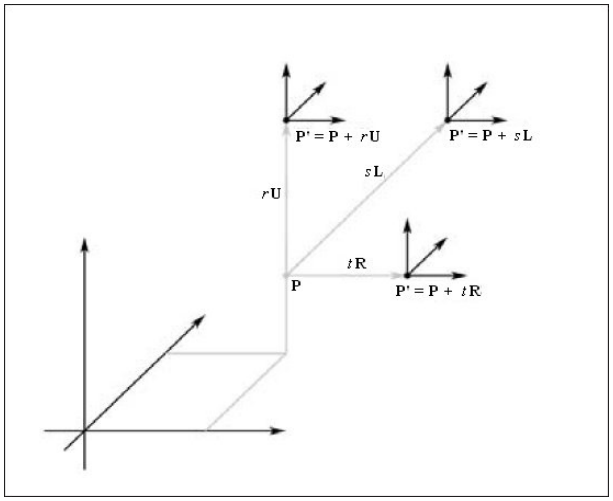

Рис. 12.7. *Перемещение вдоль векторов ориентации камеры*

Как и в случае с поворотами камеры, для наземных объектов мы вносим ряд ограничений. К примеру, объекты **`LANDOBJECT`** не могут находиться в воздухе, даже если верхний вектор изменяется в результате движения вперед или сдвига вбок. Следовательно, мы должны ограничить их перемещение плоскостью XZ. Но, поскольку наземные объекты могут изменять свою высоту, взбираясь на лестницы или холмы, мы предоставляем метод **`Camera::setPosition`**, позволяющий вручную разместить камеру в требуемом месте и на требуемой высоте.

Код реализации ходьбы, сдвига и полета выглядит следующим образом:

```
void Camera::walk(float units)
{
     // Для наземных объектов перемещение только в плоскости xz
     if(_cameraType == LANDOBJECT)
          _pos += D3DXVECTOR3(_look.x, 0.0f, _look.z) * units;
     if(_cameraType == AIRCRAFT)
          _pos += _look * units;
}
void Camera::strafe(float units)
{
     // Для наземных объектов перемещение только в плоскости xz
     if(_cameraType == LANDOBJECT)
          _pos += D3DXVECTOR3(_right.x, 0.0f, _right.z) * units;
     if(_cameraType == AIRCRAFT)
          _pos += _right * units;
}
void Camera::fly(float units)
{
     if(_cameraType == AIRCRAFT)
          _pos += _up * units;
}
```

12.3 Пример приложения: камера

Пример приложния к данной главе создает и визуализрует сцену, изображенную на рис. 12.8.

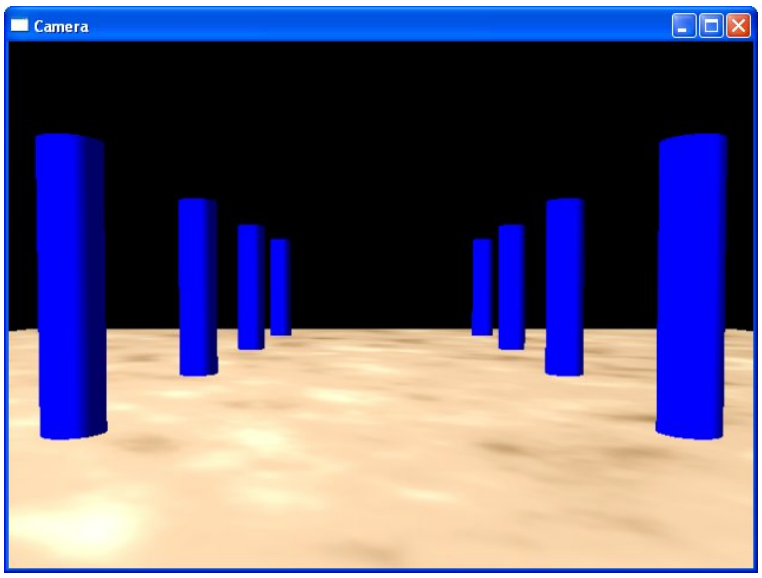

Рис. 12.8. *Окно программы, рассматриваемой в этой главе*

Вы можете свободно перемещаться по сцене, используя следующие клавиши:

- **W**/**S** — передвижение вперед и назад;
- **A**/**D** — сдвиг влево и вправо;
- **R**/**F** — полет вверх и вниз;
- Стрелки вверх/вниз — наклон камеры;
- Стрелки влево/вправо — рыскание камеры;
- **N**/**M** — вращение камеры.

Реализация примера тривиальна, поскольку вся работа выполняется внутри класса **`Camera`**, о котором мы уже говорили. В функции **`Display`** мы обрабатываем нажатие клавиш согласно их назначению. Учтите, что мы в начале программы создаем глобальный объект камеры **`TheCamera`**. Также обратите внимание, что перемещение камеры синхронизируется по прошедшему с прошлого кадра времени (**`timeDelta`**); благодаря этому скорость перемещения не зависит от частоты кадров.

```
bool Display(float timeDelta)
{
     if(Device)
     {
          //
          // Обновление сцены: перемещение камеры
          //
          if(::GetAsyncKeyState('W') & 0x8000f)
               TheCamera.walk(4.0f * timeDelta);
          if(::GetAsyncKeyState('S') & 0x8000f)
               TheCamera.walk(-4.0f * timeDelta);
          if(::GetAsyncKeyState('A') & 0x8000f)
               TheCamera.strafe(-4.0f * timeDelta);
          if(::GetAsyncKeyState('D') & 0x8000f)
               TheCamera.strafe(4.0f * timeDelta);
          if(::GetAsyncKeyState('R') & 0x8000f)
               TheCamera.fly(4.0f * timeDelta);
          if(::GetAsyncKeyState('F') & 0x8000f)
               TheCamera.fly(-4.0f * timeDelta);
          if(::GetAsyncKeyState(VK_UP) & 0x8000f)
               TheCamera.pitch(1.0f * timeDelta);
          if(::GetAsyncKeyState(VK_DOWN) & 0x8000f)
               TheCamera.pitch(-1.0f * timeDelta);
          if(::GetAsyncKeyState(VK_LEFT) & 0x8000f)
               TheCamera.yaw(-1.0f * timeDelta);
          if(::GetAsyncKeyState(VK_RIGHT) & 0x8000f)
               TheCamera.yaw(1.0f * timeDelta);
          if(::GetAsyncKeyState('N') & 0x8000f)
               TheCamera.roll(1.0f * timeDelta);
          if(::GetAsyncKeyState('M') & 0x8000f)
               TheCamera.roll(-1.0f * timeDelta);
```

```
            // Обновление матрицы вида согласно новому
            // местоположению и ориентации камеры
            D3DXMATRIX V;
            TheCamera.getViewMatrix(&V);
            Device->SetTransform(D3DTS_VIEW, &V);

            //
            // Визуализация
            //
            Device->Clear(0, 0,
                          D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                          0x00000000, 1.0f, 0);
            Device->BeginScene();

            d3d::DrawBasicScene(Device, 1.0f);

            Device->EndScene();
            Device->Present(0, 0, 0, 0);
        }
        return true;
}
```

ПРИМЕЧАНИЕ

Мы добавили в пространство имен **d3d** новую функцию **DrawBasicScene**. Эта функция рисует сцену, изображенную на рис. 12.8. Мы добавили ее к пространству имен **d3d** потому что удобно иметь функцию, рисующую стандартную сцену, благодаря чему в следующих примерах мы сможем сосредоточиться на относящемся к изучаемым темам коде и не отвлекаться на код, предназначенный для рисования сцены. Объявление в файле d3dUtility.h выглядит так:

```
// Внутри функции есть ссылка на файл desert.bmp
// Этот файл должен быть в каталоге приложения
bool DrawBasicScene(
     IDirect3DDevice9* device, // 0 для очистки
     float scale);             // масштаб
```

Когда эта функция вызывается с корректным указателем на устройство в первый раз, она выполняет инициализацию внутренних данных о геометрии сцены; поэтому рекомендуем вам первый раз вызывать эту функцию из функции **Setup**. Чтобы очистить внутреннюю геометрию, вызовите эту функцию из процедуры **Cleanup**, но вместо указателя на устройство передайте **null**. Поскольку эта функция не выполняет никаких действий, которые бы мы не обсуждали в предыдущих главах, мы предлагаем вам самостоятельно исследовать ее код, который вы найдете в сопроводительных файлах к данной главе. Обратите внимание, что в качестве текстуры функция загружает файл desert.bmp. Этот файл должен находиться в папке приложения.

12.4 Итоги

- Мы описываем местоположение и ориентацию камеры в мировой системе координат с помощью четырех векторов: правого вектора, верхнего вектора, вектора взгляда и вектора местоположения. Такое описание позволяет легко реализовать камеру с шестью степенями свободы, предоставляющую гибкий интерфейс подходящий для авиационных симуляторов и для игр с видом от первого лица.

Глава 13
Основы визуализации ландшафтов

Сетка ландшафта представляет собой обычную сетку с треугольными ячейками, подобную показанной на рис. 13.1(а), у которой для каждой вершины сетки высота задана таким образом, чтобы сетка моделировала плавный переход от гор к долинам, подобный природному ландшафту (рис. 13.1(б)). И, конечно же, мы накладываем тщательно подобранные текстуры, изображающие песчанные пляжи, покрытые травой склоны и заснеженные вершины (рис. 13.1(в)).

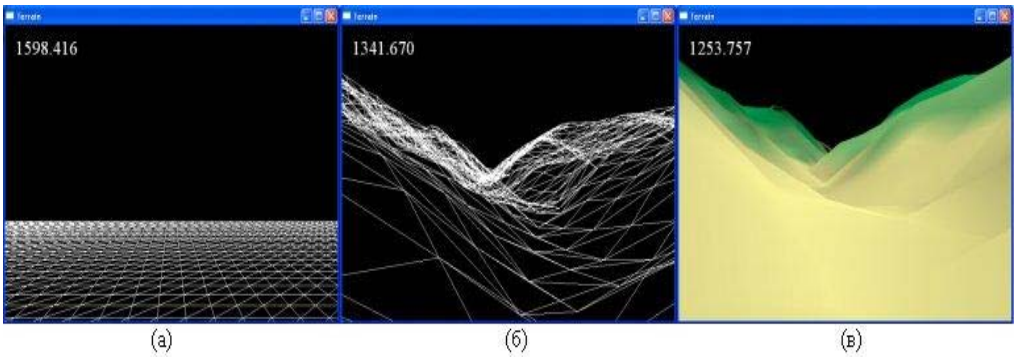

Рис. 13.1. *(а) Сетка с треугольными ячейками. (б) Сетка с плавным изменением высот. (в) Освещенный и текстурированный ландшафт, который будет формировать пример приложния, создаваемый в данной главе*

Эта глава познакомит вас с реализацией класса **`Terrain`**, использующим метод грубой силы. Под этим подразумевается, что он просто хранит данные вершин и индексов для всего ландшафта и визуализирует их. Для игр с ландшафтом небольшого размера такой подход работает, при том условии, что в компьютере установлена современная видеокарта поддерживающая аппаратную обработку вершин. Однако для тех игр, которым требуется большой ландшафт, мы должны выполнить дополнительную работу по изменению уровня детализации или отбрасыванию невидимых граней, поскольку при использовании метода грубой силы огромное количество геометрических данных, необходимых для моделирования обширных ландшафтов приведет к перегрузке видеокарты.

Цели

- Узнать как генерируется информация о высотах ландшафта, обеспечивающая плавный переход от гор к долинам, подобный природным ландшафтам.
- Изучить способы генерации данных о вершинах и треугольных гранях ландшафта.
- Познакомиться с техникой, применяемой для освещения и текстурирования ландшафта.
- Изучить способ перемещения камеры по ландшафту, чтобы создавалось впечатление ходьбы или бега.

13.1 Карты высот

Мы используем карты высот, чтобы указать, где на нашем ландшафте находятся горы, а где — долины. *Карта высот* (*heightmap*) — это массив, в котором каждый элемент задает высоту отдельной вершины сетки ландшафта. (Существует и альтернативная реализация, в которой элемент карты высот соответствует квадрату ландшафта.) Обычно мы представляем карту высот в виде матрицы, каждый элемент которой однозначно соответствует вершине сетки ландшафта.

Когда мы сохраняем карту высот на диске, для каждого ее элемента обычно отводится один байт, так что значения высоты могут находиться в диапазоне от 0 до 255. Хотя диапазон от 0 до 255 достаточен для хранения перепадов высот нашего ландшафта, в приложении может потребоваться масштабирование этого значения, чтобы оно соответствовало масштабу нашего трехмерного мира. Предположим, что в качестве единиц измерения для нашего трехмерного мира мы выбрали сантиметры, тогда диапазона значений от 0 до 255 будет недостаточно, чтобы изобразить что-нибудь интересное. Поэтому при загрузке данных в приложение мы выделяем для каждого элемента карты высот переменную типа **`int`** или **`float`**. Это позволяет масштабировать значения за пределы диапазона от 0 до 255, чтобы они соответствовали любому требуемому масштабу.

Одним из возможных представлений карты высот является карта с градациями серого, где темные области соответствуют низинам, а светлые — возвышенностям. Такая карта показана на рис. 13.2.

Рис. 13.2. *Карта высот в виде градаций серого*

13.1.1 Создание карты высот

Карта высот может быть создана либо программно, либо с помощью графического редактора, например Adobe Photoshop. Использование графического редактора — более простой способ, позволяющий интерактивно создавать ландшафты и видеть полученный результат. Кроме того, можно создавать необычные карты высот, пользуясь дополнительными возможностями графического редактора, такими как фильтры. На рис. 13.3 показана пирамидальная карта высот, созданная в Adobe Photoshop с помощью инструментов редактирования. Обратите внимание, что при создании изображения мы выбираем карту градаций серого.

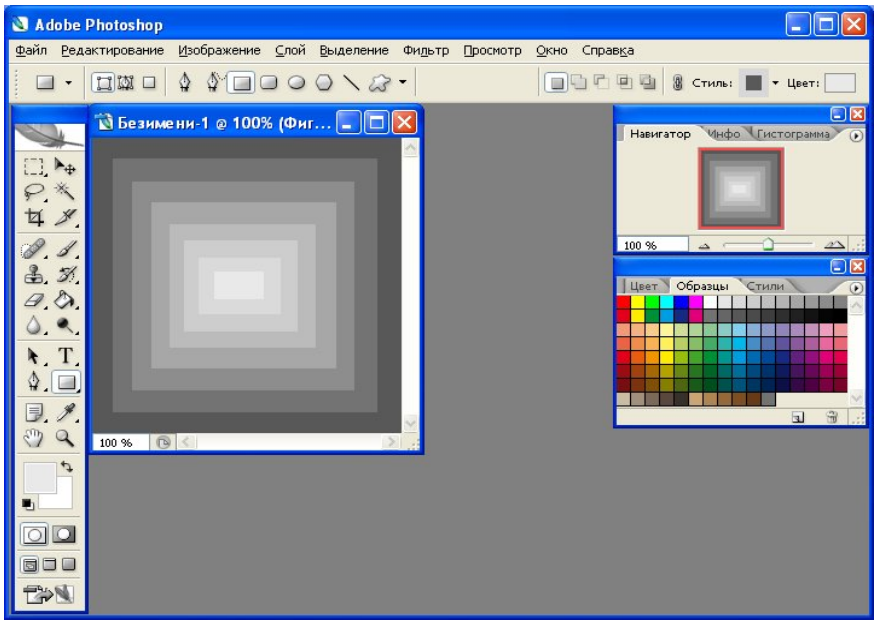

Рис. 13.3. *Карта градаций серого, создднная в Adobe Photoshop*

Когда вы завершите рисование своей карты высот, сохраните ее как 8-разрядный файл RAW. Файлы RAW просто содержат байты изображения один за другим. Благодаря этому чтение данных изображения в приложении выполняется очень просто. Ваш графический редактор может спростить, сохранять ли изображение в файл RAW с заголовком. Укажите, что заголовок не нужен.

ПРИМЕЧАНИЕ Вы не обязаны использовать для хранения информации о высотах формат RAW; применяйте любой формат, соответствующий вашим потребностям. Формат RAW — это только один пример формата, который можно использовать. Мы выбрали его по той причине, что большинство популярных графических редакторов могут выполнять экспорт изображения в этот формат и чтение данных из файла RAW в приложении реализуется очень просто. В примерах из этой главы используются 8-разрядные файлы RAW.

13.1.2 Загрузка файла RAW

Поскольку файл RAW — это всего лишь непрерывный массив байтов, мы можем просто прочитать его целиком с помощью приведенного ниже метода. Обратите внимание, что переменная **`_heightmap`** — это член класса **`Terrain`**, определеный следующим образом: **`std::vector<int> _heightmap;`**.

```
bool Terrain::readRawFile(std::string fileName)
{
     // Высота для каждой вершины
     std::vector<BYTE> in(_numVertices);
```

```
     std::ifstream inFile(fileName.c_str(), std::ios_base::binary);

     if(inFile == 0)
          return false;

     inFile.read(
          (char*)&in[0], // буффер
          in.size());    // количество читаемых в буфер байт

     inFile.close();

     // копируем вектор BYTE в вектор int
     _heightmap.resize(_numVertices);
     for(int i = 0; i < in.size(); i++)
          _heightmap[i] = in[i];

     return true;
}
```

Обратите внимание, что мы копируем вектор байтов в вектор целых чисел; это делается для того, чтобы потом мы могли масштабировать значения высот для выхода за пределы диапазона [0, 255].

Единственным ограничением данного метода является то, что количество байт в читаемом файле RAW должно быть не меньше количества вершин в сетке ландшафта. Следовательно, если вы считываете файл RAW размером 256 × 256, то должны создать ландшафт в котором будет не более 256 × 256 вершин.

13.1.3 Досуп к карте высот и ее модификация

Для доступа к элементам карты высот и их модификации класс **`Terrain`** предоставляет следующие два метода:

```
int Terrain::getHeightmapEntry(int row, int col)
{
     return _heightmap[row * _numVertsPerRow + col];
}

void Terrain::setHeightmapEntry(int row, int col, int value)
{
     _heightmap[row * _numVertsPerRow + col] = value;
}
```

Эти методы позволяют ссылаться на элемент карты, указывая номера строки и столбца, и скрывают выполняемое преобразование в индекс одномерного массива.

13.2 Создание геометрии ландшафта

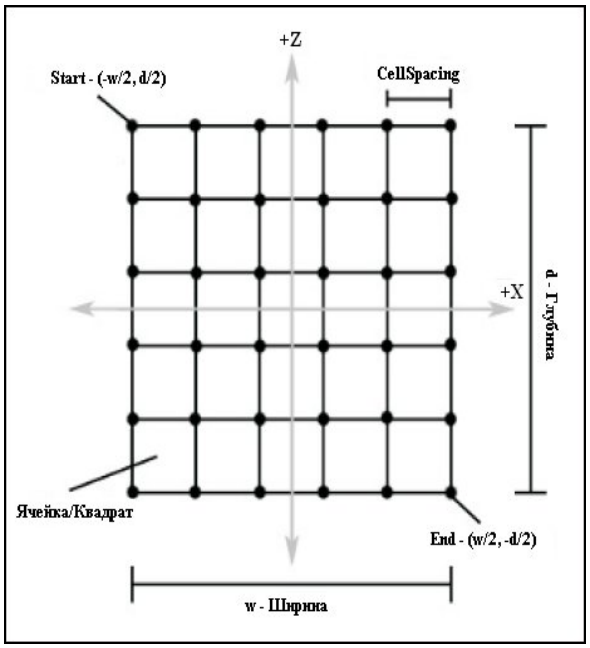

Рис. 13.4. *Свойства размеченной треугольной сетки. Точки на пересечении линий сетки обозначают вершины*

Рис. 13.4 иллюстрирует свойства ландшафта, термины и специальные точки, на которые мы будем ссылаться. Размер ландшафта определяется путем указания количества вершин в строке, количества вершин в столбце и размера ячейки. Все эти значения мы передаем в класс **`Terrain`**. Кроме того, мы передаем указатель на связанное с ландшафтом устройство, строку с именем файла, содержащего карту высот и коэффициент масштабирования, используемый при масштабировании элементов карты высот.

```
class Terrain
{
public:
     Terrain(
          IDirect3DDevice9* device,
          std::string heightmapFileName,
          int numVertsPerRow,
          int numVertsPerCol,
          int cellSpacing,    // расстояние между вершинами
          float heightScale); // коэффициент масштабирования высоты

     ... методы пропущены

private:
     ... устройство, буфер вершин и т.п. пропущены

     int _numVertsPerRow;
     int _numVertsPerCol;
```

```
     int _cellSpacing;
     int _numCellsPerRow;
     int _numCellsPerCol;
     int _width;
     int _depth;
     int _numVertices;
     int _numTriangles;
     float _heightScale;
};
```

Чтобы увидеть объявление класса **Terrain** полностью, посмотрите исходный код примера в сопроводительных файлах; оно слишком велико, чтобы приводить его в тексте.

На основании переданных конструктору данных мы вычисляем другие параметры ландшафта:

```
_numCellsPerRow = _numVertsPerRow - 1;
_numCellsPerCol = _numVertsPerCol - 1;
_width = _numCellsPerRow * _cellSpacing;
_depth = _numCellsPerCol * _cellSpacing;
_numVertices = _numVertsPerRow * _numVertsPerCol;
_numTriangles = _numCellsPerRow * _numCellsPerCol * 2;
```

Кроме того, мы объявляем структуру вершин ландшафта следующим образом:

```
struct TerrainVertex
{
     TerrainVertex(){}
     TerrainVertex(float x, float y, float z, float u, float v)
     {
          _x = x; _y = y; _z = z; _u = u; _v = v;
     }
     float _x, _y, _z;
     float _u, _v;

     static const DWORD FVF;
};
const DWORD Terrain::TerrainVertex::FVF = D3DFVF_XYZ | D3DFVF_TEX1;
```

Обратите внимание, что **TerrainVertex** — это вложенный класс, объявленный внутри класса **Terrain**. Мы выбрали такую реализацию по той причине, что класс **TerrainVertex** не нужен вне класса **Terrain**.

13.2.1 Вычисление вершин

Во время последующего обсуждения смотрите на рис. 13.4. Чтобы создать вершины нашей сетки мы просто начинаем генерировать данные вершин с той, которая отмечена на рисунке словом *start* и заполняем вершины ряд за рядом, пока не дойдем до той, которая отмечена словом *end*; при этом расстояние между вершинами задается параметром *cellSpacing*. Это позволяет нам получить координаты X и Z, но как насчет координаты Y? Координата Y берется из соответствующего элемента загруженной карты высот.

ПРИМЕЧАНИЕ Рассматриваемая реализация использует один большой буфер вершин для хранения данных всех вершин всего ландшафта. Это может вызвать проблемы, связанные с накладываемым оборудованием ограничениями. Например, параметры устройства задают максимальное количество примитивов и максимальное количество индексов вершин. Проверьте значения членов **`MaxPrimitiveCount`** и **`MaxVertexIndex`** структуры **`D3DCAPS9`**, чтобы увидеть какие ограничения накладывает используемое вами оборудование. Решение проблем ,вызванных использованием одного буфера вершин, обсуждается в разделе 13.7.

При вычислении координат текстуры, учитывая рис. 13.5, получаем простой сценарий, определяющий соответствие координат текстуры (u, v) вершине ландшафта (i, j) по следующей формуле:

$$u = j \cdot uCoordIncrementSize$$
$$v = i \cdot uCoordIncrementSize$$

где

$$uCoordIncrementSize = \frac{1}{numCellCols}$$
$$vCoordIncrementSize = \frac{1}{numCellRows}$$

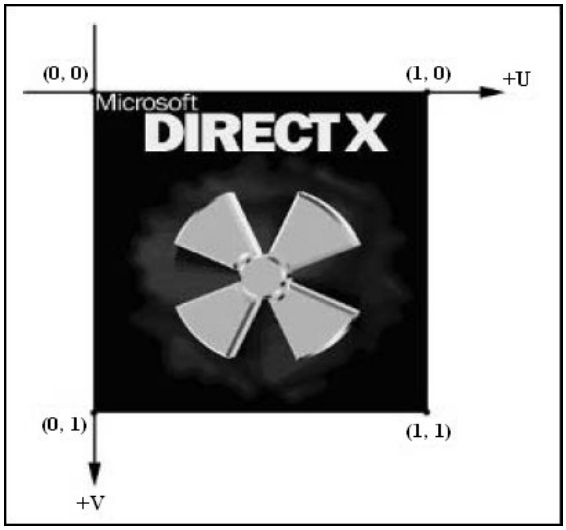

Рис. 13.5. *Соответствие между вершинами ландшафта и координатами текстур*

И, в заключение, приведем код генерации данных вершин:

```
bool Terrain::computeVertices()
{
     HRESULT hr = 0;
```

```
    hr = _device->CreateVertexBuffer(
                    _numVertices * sizeof(TerrainVertex),
                    D3DUSAGE_WRITEONLY,
                    TerrainVertex::FVF,
                    D3DPOOL_MANAGED,
                    &_vb,
                    0);

    if(FAILED(hr))
         return false;

    // координаты, с которых начинается генерация вершин
    int startX = -_width / 2;
    int startZ =  _depth / 2;

    // координаты, на которых завершается генерация вершин
    int endX =  _width / 2;
    int endZ = -_depth / 2;

    // вычисляем приращение координат текстуры
    // при переходе от одной вершины к другой.
    float uCoordIncrementSize = 1.0f / (float)_numCellsPerRow;
    float vCoordIncrementSize = 1.0f / (float)_numCellsPerCol;

    TerrainVertex* v = 0;
    _vb->Lock(0, 0, (void**)&v, 0);

    int i = 0;
    for(int z = startZ; z >= endZ; z -= _cellSpacing)
    {
         int j = 0;
         for(int x = startX; x <= endX; x += _cellSpacing)
         {
              // вычисляем правильный индекс в буфере вершин и
              // карте высот на основании счетчиков вложенных циклов
              int index = i * _numVertsPerRow + j;

              v[index] = TerrainVertex(
                                  (float)x,
                                  (float)_heightmap[index],
                                  (float)z,
                                  (float)j * uCoordIncrementSize,
                                  (float)i * vCoordIncrementSize);

              j++; // следующий столбец
         }
         i++; // следующая строка
    }

    _vb->Unlock();

    return true;
}
```

13.2.2 Вычисление индексов — определение треугольников

Чтобы вычислить индексы сетки мы просто в цикле перебираем все ее квадраты, начиная с верхнего левого и заканчивая правым нижним, как показано на рис. 13.4, и формируем два треугольника, образующие квадрат.

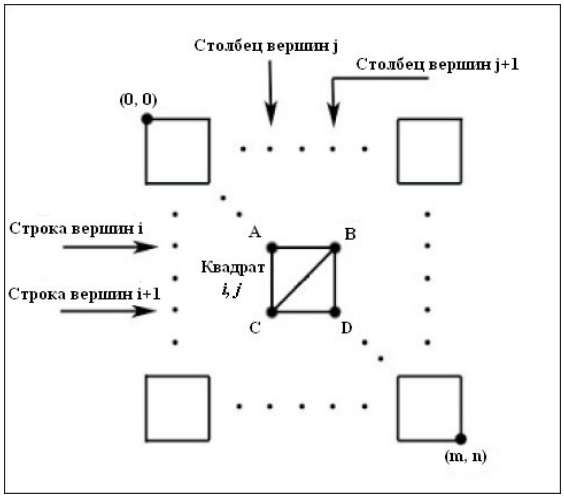

Рис. 13.6. *Вершины квадрата*

Главным трюком здесь является общая формула, позволяющая вычислить индексы треугольников, образующих квадрат в позиции (i, j). Рис. 13.6 позволяет вывести общую формулу для квадрата(i, j):

```
ΔABC = { i*numVertsPerRow + j,
         i*numVertsPerRow + j + 1,
         (i + 1)*numVertsPerRow + j }

ΔCBD = { (i + 1)*numVertsPerRow + j,
         i*numVertsPerRow + j + 1,
         (i + 1)*numVertsPerRow + j + 1 }
```

Вот код генерации индексов:

```
bool Terrain::computeIndices()
{
     HRESULT hr = 0;

     hr = _device->CreateIndexBuffer(
            _numTriangles * 3 * sizeof(WORD), // 3 индекса на
                                              // треугольник
            D3DUSAGE_WRITEONLY,
            D3DFMT_INDEX16,
            D3DPOOL_MANAGED,
            &_ib,
            0);
```

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

    WORD* indices = 0;
    _ib->Lock(0, 0, (void**)&indices, 0);

    // Индекс, с которого начинается группа из 6 индексов,
    // описывающая два треугольника, образующих квадрат
    int baseIndex = 0;

    // В цикле вычисляем треугольники для каждого квадрата
    for(int i = 0; i < _numCellsPerCol; i++)
    {
       for(int j = 0; j < _numCellsPerRow; j++)
       {
           indices[baseIndex]     = i * _numVertsPerRow + j;
           indices[baseIndex + 1] = i * _numVertsPerRow + j + 1;
           indices[baseIndex + 2] = (i+1) * _numVertsPerRow + j;
           indices[baseIndex + 3] = (i+1) * _numVertsPerRow + j;
           indices[baseIndex + 4] = i * _numVertsPerRow + j + 1;
           indices[baseIndex + 5] = (i+1) * _numVertsPerRow + j + 1;

              // следующий квадрат
              baseIndex += 6;
       }
    }

    _ib->Unlock();

    return true;
}
```

13.3 Текстурирование

Класс **Terrain** предоставляет два способа текстурирования ландшафта. Наиболее очевидный способ — загрузить ранее подготовленную текстуру из файла и использовать ее. Показанный ниже метод, реализованный в классе **Terrain**, загружает текстуру из файла в член данных **_tex**, являющийся указателем на интерфейс **IDirect3DTexture9**. Внутри метода **Terrain::draw** перед визуализацией ландшафта устанавливается текстура **_tex**.

Если вы прочитали предыдущие главы, реализация метода не должна вызвать у вас никаких вопросов.

```
bool Terrain::loadTexture(std::string fileName)
{
    HRESULT hr = 0;
    hr = D3DXCreateTextureFromFile(
                    _device,
                    fileName.c_str(),
                    &_tex);
```

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

     return true;
}
```

13.3.1 Процедурный подход

Альтернативным способом текстурирования ландшафта является программное вычисление текстуры; это означает, что мы создаем «пустую» текстуру и вычисляем цвет каждого ее текселя в коде на основании некоторых предопределенных параметров. В нашем примере таким параметром является высота вершины ландшафта.

Программная генерация текстуры выполняется в методе **`Terrain::genTexture`**. Сперва мы создаем пустую текстуру с помощью метода **`D3DXCreateTexture`**. Затем мы блокируем текстуру верхнего уровня (помните, что это детализируемая текстура и у нее есть несколько уровней детализации). После этого мы в цикле перебираем тексели и назначаем их цвет. Цвета текселей зависят от высоты вершин квадрата сетки, которому они принадлежат. Идея заключается в том, что низкие участки ландшафта окрашиваются в цвет песчанного пляжа, участки со средней высотой — в цвет травы, а высокие части ландшафта — в цвет снежных вершин. Мы считаем, что высота квадрата это высота его верхнего левого угла.

Назначив цвета всем текселям, мы должны сделать некоторые из них темнее или светлее в зависимости от того, под каким углом солнечный свет (моделируемый с помощью источника направленного света) падает на квадрат ландщафта, соответствующий данному текселю. Все это делается в методе **`Terrain::lightTerrain`**, реализация которого будет обсуждаться в следующием разделе.

В конце метода **`Terrain::genTexture`** осуществляется вычисление текселей остальных уровней детализируемой текстуры. Это делается с помощью функции **`D3DXFilterTexture`**. Вот как выглядит код генерации текстуры:

```
bool Terrain::genTexture(D3DXVECTOR3* directionToLight)
{
     // Метод программно заполняет текстуру верхнего уровня.
     // Затем выполняется ее освещение. И, в конце, заполняются
     // остальные уровни детализируемой текстуры с помощью
     // D3DXFilterTexture.

     HRESULT hr = 0;


     // Тексель для каждого квадрата сетки
     int texWidth  = _numCellsPerRow;
     int texHeight = _numCellsPerCol;
```

```
// Создаем пустую текстуру
hr = D3DXCreateTexture(
       _device,
       texWidth, texHeight, // размеры
       0,                   // создаем полную
                            // цепочку детализации
       0,                   // использование – нет
       D3DFMT_X8R8G8B8,     // формат 32-разрядный XRGB
       D3DPOOL_MANAGED,     // пул памяти
       &_tex);

if(FAILED(hr))
     return false;

D3DSURFACE_DESC textureDesc;
_tex->GetLevelDesc(0 /* уровень */, &textureDesc);

// Проверяем, что получена текстура требуемого формата,
// поскольку наш код заполнения текстуры работает только
// с 32-разрядными пикселями
if(textureDesc.Format != D3DFMT_X8R8G8B8)
     return false;

D3DLOCKED_RECT lockedRect;
_tex->LockRect(0,           // блокируем верхнюю поверхность
               &lockedRect,
               0,           // блокируем всю текстуру
               0);          // флаги

// Заполняем текстуру
DWORD* imageData = (DWORD*)lockedRect.pBits;
for(int i = 0; i < texHeight; i++)
{
     for(int j = 0; j < texWidth; j++)
     {
          D3DXCOLOR c;
          // Получаем высоту верхней левой вершины квадрата
          float height = (float)getHeightmapEntry(i, j) /
                         _heightScale;

          // Устанавливаем цвет текселя на основе высоты
          // соответствующего ему квадрата
          if((height) < 42.5f)      c = d3d::BEACH_SAND;
          else if((height) < 85.0f) c = d3d::LIGHT_YELLOW_GREEN;
          else if((height) < 127.5f) c = d3d::PUREGREEN;
          else if((height) < 170.0f) c = d3d::DARK_YELLOW_GREEN;
          else if((height) < 212.5f) c = d3d::DARKBROWN;
          else                       c = d3d::WHITE;


          // Заполняем заблокированный буфер. Обратите внимание,
          // что мы делим шаг на четыре, поскольку шаг
          // измеряется в байтах, а одно значение DWORD
          // занимает 4 байта
```

```
                imageData[i * lockedRect.Pitch / 4 + j] = (D3DCOLOR)c;
            }
        }
        _tex->UnlockRect(0);

        // Освещаем ландшафт
        if(!lightTerrain(directionToLight))
        {
            ::MessageBox(0, "lightTerrain() - FAILED", 0, 0);
            return false;
        }

        // Заподняем цепочку детализации
        hr = D3DXFilterTexture(
                _tex,   // текстура, для которой заполняются
                        // уровни детализации
                0,      // палитра по умолчанию
                0,      // используем в качестве источника
                        // верхний уровень
                D3DX_DEFAULT); // фильтр по умолчанию

        if(FAILED(hr))
        {
            ::MessageBox(0, "D3DXFilterTexture() - FAILED", 0, 0);
            return false;
        }
        return true;
}
```

Обратите внимание, что константы цветов, **`BEACH_SAND`** и т.п., определены в файле d3dUtility.h.

13.4 Освещение

Метод **`Terrain::genTexture`** обращается к функции **`Terrain::lightTerrain`**, которая, как говорит ее имя, выполняет освещение ландшафта для увеличения реализма сцены. Поскольку мы уже вычислили цвета текстуры ландшафта, нам осталось вычислить только коэффициент затенения, который делает отдельные участки темнее или светлее в зависимости от их расположения относительно источника света. В данном разделе мы исследуем такую технику. Вы можете недоумевать, почему мы занялись освещением ландшафта, а не позволили Direct3D все сделать за нас. У самостоятельного выполнения вычислений есть три преимущества:

- Мы экономим память, поскольку нам не надо хранить нормали вершин.
- Так как ландшафты статичны и мы не будем перемещать источники света, можно заранее рассчитать освещение, освободив то время, которое Direct3D тратил бы на расчет освещения ландшафта в реальном времени.
- Мы попрактикуемся в математике, познакомимся с базовыми концепциями освещения и поработаем с функциями Direct3D.

13.4.1 Обзор

Техника освещения, которую мы будем использовать для вычисления оттенков ландшафта является одной из самых простых и известна как *рассеянное освещение* (*diffuse lighting*). Мы используем параллельный источник света, который описываем путем указания направления на источник света, являющегося противоположным тому направлению, в котором падают испускаемые источником лучи. Например, если мы хотим чтобы лучи света падали с неба вертикально вниз в направлении *lightRaysDirection* = (0, –1, 0), то направлением на источник света будет указывающий в противоположном направлении вектор *directionToLight* = (0, 1, 0). Обратите внимание, что мы используем единичные векторы.

ПРИМЕЧАНИЕ Хотя указание направления испускаемых источником света лучей может казаться более интуитивно понятным, задание направления на источник света больше подходит для вычисления рассеянного освещения.

Затем для каждого квадрата ландшафта мы вычисляем угол между вектором освещения $\hat{\mathbf{L}}$ и нормалью к поверхности квадрата $\hat{\mathbf{N}}$.

На рис. 13.7 видно, что чем больше угол, тем больше поверхность квадрата отворачивается от источника света и тем меньше света попадает на поверхность. Соответственно, чем меньше угол, тем больше поверхность квадрата повернута к источнику света и тем больше света она получает. Кроме того, обратите внимание, что если угол между вектором освещения и нормалью поверхности больше 90 градусов, поверхность вообще не освещена.

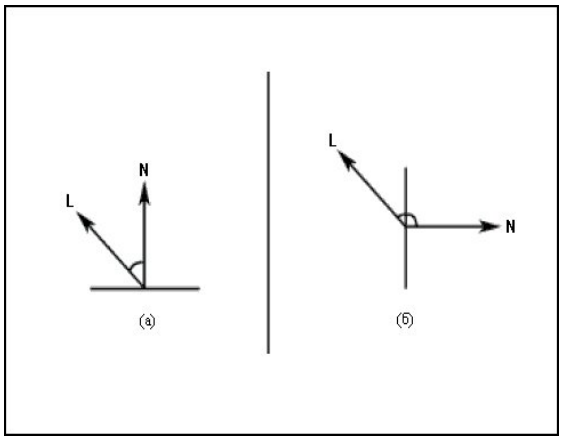

Рис. 13.7. *Угол между вектором освещения* $\hat{\mathbf{L}}$ *и нормалью поверхности* $\hat{\mathbf{N}}$ *определяет сколько света получает поверхность. На рисунке (а) угол меньше 90 градусов. На рисунке (б) угол больше 90 градусов. Заметьте, что во втором случае поверхность не получает света потому что лучи света (испускаемые в направлении противоположном вектору* $\hat{\mathbf{L}}$*) попадают на обратную сторону поверхности*

Используя угловые отношения между вектором освещения и нормалью поверхности можно вычислить коэффициент затенения, находящийся в диапазоне [0, 1], который определяет сколько света получает поверхность. Большие углы представляются близкими к нулю значениями коэффициента. Когда цвет умножается на близкий к нулю коэффициент затенения, он становится темнее, а это именно то, что нам надо. С другой стороны, малые углы представляются

близкими к единице значениями коэффициента, и умножение на этот коэффициент практически не меняет яркость цвета.

13.4.2 Вычисление затенения квадрата

Направление на источник света нам дано в виде нормализованного вектора $\hat{\mathbf{L}}$. Чтобы вычислить угол между $\hat{\mathbf{L}}$ и нормалью квадрата $\hat{\mathbf{N}}$ нам сперва необходимо получить $\hat{\mathbf{N}}$. Это тривиальная задача, решаемая с помощью векторного произведения. Но сначала мы должны получить два ненулевых и непараллельных вектора, лежащих в той же плоскости, что и квадрат. На рис. 13.8 эти два вектора обозначены **u** и **v**:

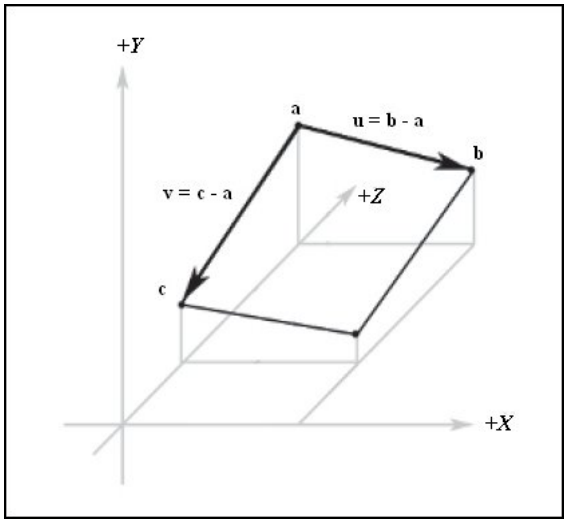

Рис. 13.8. *Вычисление двух векторов, находящихся в одной плоскости с квадратом*

$$\mathbf{u} = \left(cellSpacing, \quad b_y - a_y, \quad 0\right)$$
$$\mathbf{v} = \left(0, \quad c_y - a_y, \quad -cellSpacing\right)$$

Получив **u** и **v**, нормаль квадрата **N** вычисляем по формуле $\mathbf{N} = \mathbf{u} \times \mathbf{v}$. Конечно же следует нормализовать вектор **N**:

$$\hat{\mathbf{N}} = \frac{\mathbf{N}}{|\mathbf{N}|}$$

Чтобы найти угол между $\hat{\mathbf{L}}$ и $\hat{\mathbf{N}}$ вспомним, что скалярное произведение двух единичных векторов в трехмерном пространстве равно косинусу угла между ними:

$$\hat{\mathbf{L}} \cdot \hat{\mathbf{N}} = s$$

Значения коэффициента *s* будут находиться в интервале [–1, 1]. Значения *s* из диапазона [–1, 0) соответствуют углам между $\hat{\mathbf{L}}$ и $\hat{\mathbf{N}}$ большим 90 градусов, а в

этом случае, как показано на рис. 13.7 поверхность не получает света. Поэтому любые значения из диапазона [–1, 0) мы заменяем на 0:

```
float cosine = D3DXVec3Dot(&n, directionToLight);

if(cosine < 0.0f)
     cosine = 0.0f;
```

Теперь, когда значения *s* для углов больших 90 градусов отброшены, *s* становится нашим коэффициентом затенения с интервалом значений [0, 1], поскольку когда угол между $\hat{L}$ и $\hat{N}$ увеличивается от 0 до 90 градусов значение s изменяется от 1 до 0. Это именно та необходимая нам функциональность о которой говорилось в разделе 13.4.1.

Коэффициент затенения для отдельного квадрата вычисляет метод **`Terrain::computeShade`**. В качестве параметров он получает номера строки и столбца, идентифицирующие квадрат ландшафта и вектор, задающий направление на параллельный источник света.

```
float Terrain::computeShade(int cellRow, int cellCol,
                            D3DXVECTOR3* directionToLight)
{
     // Получаем высоты трех вершин квадрата
     float heightA = getHeightmapEntry(cellRow,   cellCol);
     float heightB = getHeightmapEntry(cellRow,   cellCol+1);
     float heightC = getHeightmapEntry(cellRow+1, cellCol);

     // Строим два вектора квадрата
     D3DXVECTOR3 u(_cellSpacing, heightB - heightA, 0.0f);
     D3DXVECTOR3 v(0.0f, heightC - heightA, -_cellSpacing);

     // Находим нормаль, выполнив векторное умножение
     // двух векторов квадрата
     D3DXVECTOR3 n;
     D3DXVec3Cross(&n, &u, &v);
     D3DXVec3Normalize(&n, &n);

     float cosine = D3DXVec3Dot(&n, directionToLight);

     if(cosine < 0.0f)
          cosine = 0.0f;

     return cosine;
}
```

13.4.3 Затенение ландшафта

Теперь, когда мы узнали как выполнить затенение отдельного квадрата, можно выполнить затенение всех квадратов ландшафта. Мы просто перебираем в цикле все квадраты ландшафта, вычисляем для каждого из них коэффициент затенения и умножаем цвет соответствующего квадрату текселя на полученный коэффициент. В результате квадраты, получающие мало света станут темнее. Ниже приведен

фрагмент кода, являющийся главной частью метода **Terrain::lightTerrain**:

```
DWORD* imageData = (DWORD*)lockedRect.pBits;
for(int i = 0; i < textureDesc.Height; i++)
{
     for(int j = 0; j < textureDesc.Width; j++)
     {
          int index = i * lockedRect.Pitch / 4 + j;

          // Получаем текущий цвет ячейки
          D3DXCOLOR c(imageData[index]);

          // Затеняем текущую ячейку
          c *= computeShade(i, j, lightDirection);

          // Сохраняем затененный цвет
          imageData[index] = (D3DCOLOR)c;
     }
}
```

13.5 «Ходьба» по ландшафту

После того, как мы сконструировали ландшафт, хорошо добавить возможность перемещать камеру таким образом, чтобы она имитировала ходьбу по ландшафту. То есть нам надо менять высоту камеры (координату Y) в зависимости от того, в каком месте ландшафта мы находимся. Для этого мы сначала должны определить по координатам X и Z камеры квадрат ландшафта в котором мы находимся. Все это делает функция **Terrain::getHeight**; в своих параметрах она получает координаты X и Z камеры и возвращает высоту, на которой должна быть расположена камера, чтобы она оказалась над ландшафтом. Давайте рассмотрим реализацию функции.

```
float Terrain::getHeight(float x, float z)
{
     // Выполняем преобразование перемещения для плоскости XZ,
     // чтобы точка START ландшафта совпала с началом координат.
     x = ((float)_width / 2.0f) + x;
     z = ((float)_depth / 2.0f) - z;

     // Масштабируем сетку таким образом, чтобы размер каждой ее
     // ячейки стал равен 1. Для этого используем коэффициент
     // 1 / cellspacing поскольку cellspacing * 1 / cellspacing = 1.
     x /= (float)_cellSpacing;
     z /= (float)_cellSpacing;
```

Сперва мы выполняем перемещение в результате которого начальная точка ландшафта будет совпадать с началом координат. Затем мы выполняем операцию масштабирования с коэффициентом равным единице деленной на размер клетки; в результате размер клетки ландшафта будет равен 1. Затем мы переходим к новой системе координат, где положительное направление оси Z направлено «вниз».

Конечно, вы не найдете кода меняющего систему координат, просто помните, что ось Z направлена вниз. Все эти этапы показаны на рис. 13.9.

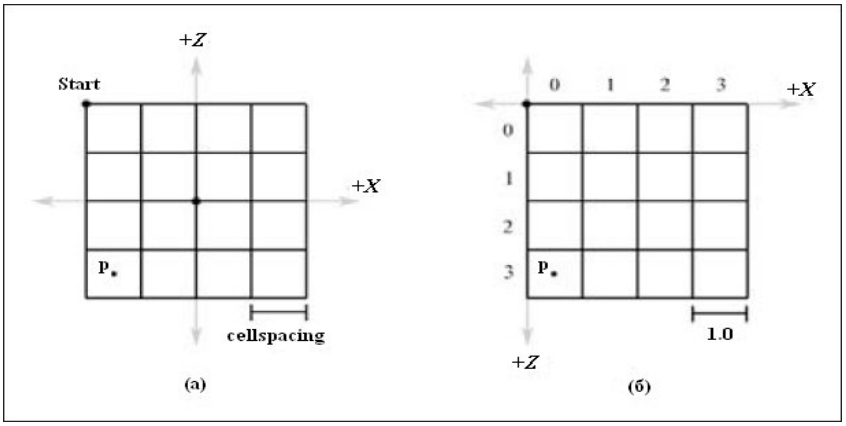

Рис. 13.9. *Исходная сетка ландшафта и сетка после переноса начальной точки ландшафта в начало координат, масштабирования, делающего размер квадрата равным 1 и смены направления оси Z*

Мы видим, что измененная координатная система соответствует порядку элементов матрицы. То есть верхний левый угол — это начало координат, номера столбцов увеличиваются вправо, а номера строк растут вниз. Следовательно, согласно рис 13.9, и помня о том, что размер ячейки равен 1, мы сразу видим что номер строки и столбца для той клетки на которой мы находимся вычисляется так:

```
float col = ::floorf(x);
float row = ::floorf(z);
```

Другими словами, номер столбца равен целой части координаты X, а номер строки — целой части координаты Z. Также вспомните, что результатом функции **`floor(t)`** является наибольшее целое число, которое меньше или равно **`t`**.

Теперь, когда мы знаем в какой ячейке находимся, можно получить высоты ее четырех вершин:

```
// A   B
// *---*
// | / |
// *---*
// C   D
float A = getHeightmapEntry(row,   col);
float B = getHeightmapEntry(row,   col+1);
float C = getHeightmapEntry(row+1, col);
float D = getHeightmapEntry(row+1, col+1);
```

Теперь мы знаем в какой ячейке находимся и высоты всех четырех ее вершин. Нам надо найти высоту (координату Y) точки ячейки с указанными координатами X и Z, где находится камера. Это нетривиальная задача, поскольку ячейка может быть наклонена, как показано на рис. 13.10.

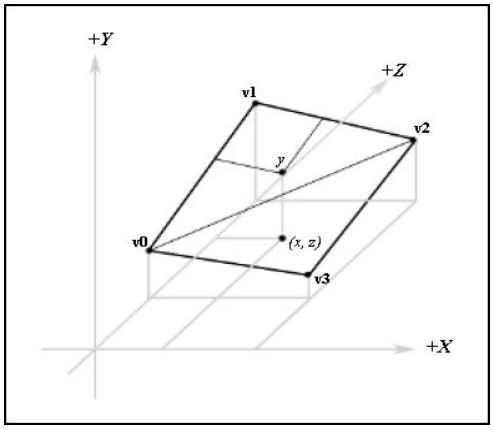

Рис. 13.10. *Высота ячейки (координата Y) для заданных координат местоположения камеры X и Z*

Чтобы определить высоту мы должны узнать в каком треугольнике ячейки мы находимся. Вспомните, что каждая ячейка визуализируется как два треугольника. Чтобы определить в каком треугольнике мы находимся, мы берем тот квадрат сетки в котором находимся и перемещаем его таким образом, чтобы верхняя левая вершина совпадала с началом координат.

Поскольку переменные **`row`** и **`col`** определяют местоположение левой верхней вершины той ячейки где мы находимся, необходимо выполнить перемещение на **`-col`** по оси X и **`-row`** по оси Z. Преобразование координат X и Z выполняется так:

```
float dx = x - col;
float dz = z - row;
```

Ячейка после выполнения преобразования показана на рис. 13.11.

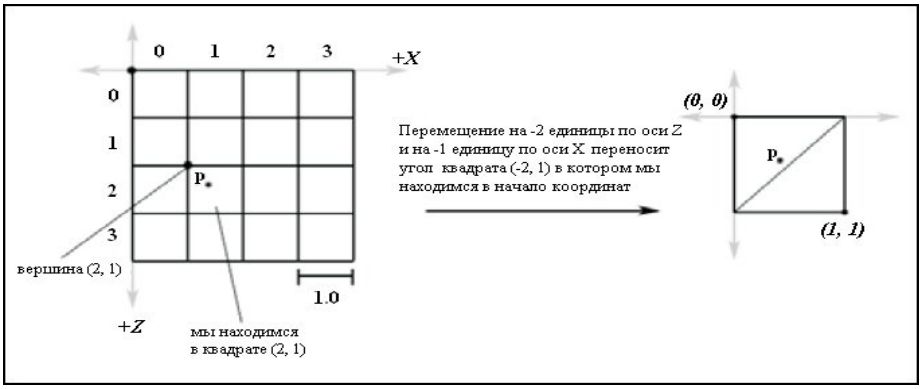

Рис. 13.11. *Ячейка до и после преобразования, переносящего ее верхнюю левую вершину в начало координат*

Теперь, если $dz < 1.0 - dx$ мы находимся в «верхнем» треугольнике **`Δv0v1v2`**. В ином случае мы находимся в «нижнем» треугольнике **`Δv0v2v3`** (рис. 13.10).

Теперь мы посмотрим как определить высоту, если мы находимся в «верхнем» треугольнике. Для «нижнего» треугольника процесс аналогичен и ниже будет

приведен код для обоих вариантов. Чтобы найти высоту когда мы находимся в «верхнем» треугольнике, надо сформировать два вектора $\mathbf{u} = (cellSpacing, B - A, 0)$ и $\mathbf{v} = (0, C - A, -cellSpacing)$, совпадающих со сторонами треугольника и начинающихся в точке, заданной вектором $\mathbf{q} = (q_x, A, q_z)$, как показано на рис. 13.12(а) Затем мы выполняем линейную интерполяцию вдоль вектора **u** на *dx* и вдоль вектора **v** на *dz*. Эти интерполяции показаны на рис. 13.12(б). Координата Y вектора ($\mathbf{q} + dx\mathbf{u} + dz\mathbf{v}$) дает нам высоту для заданных координат X и Z; чтобы увидеть, как это происходит, вспомните геометрическую интерпретацию сложения векторов.

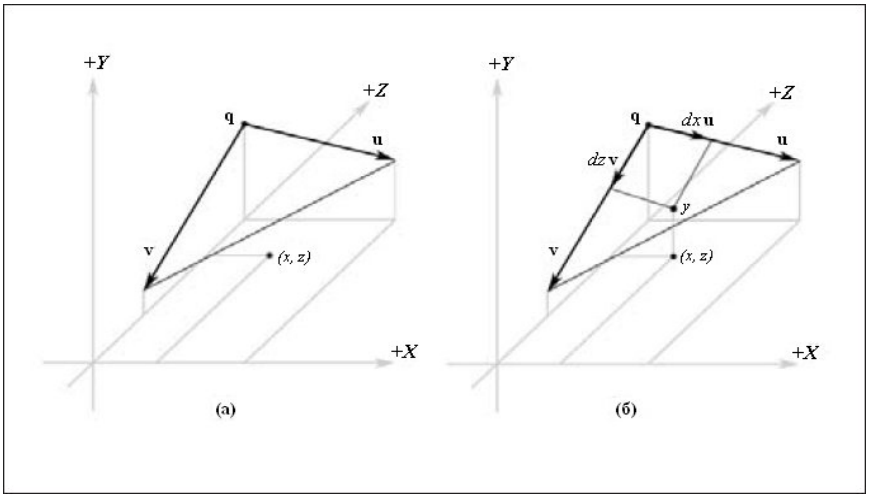

Рис. 13.12. *(а) Вычисляем два вектора, совпадающих со сторонами треугольника. (б) Высота вычисляется путем линейной интерполяции* ***u*** *на dx и* ***v*** *на dz*

Обратите внимание, что поскольку нас интересует только значение высоты, мы можем выполнять интерполяцию только для компоненты *y* и игнорировать остальные компоненты. В этом случае высота определяется по формуле $A + dxu_y + dzv_y$.

Итак, вот заключительная часть кода метода **`Terrian::getHeight`**:

```
if(dz < 1.0f - dx) // верхний треугольник ABC
{
     float uy = B - A; // A->B
     float vy = C - A; // A->C


     height = A + d3d::Lerp(0.0f, uy, dx) +
                  d3d::Lerp(0.0f, vy, dz);
}
else // нижний треугольник DCB
{
     float uy = C - D; // D->C
     float vy = B - D; // D->B
```

```
     height = D + d3d::Lerp(0.0f, uy, 1.0f - dx) +
                  d3d::Lerp(0.0f, vy, 1.0f - dz);
}
return height;
}
```

Функция **Lerp** выполняет линейную интерполяцию вдоль одномерной линии и ее реализация выглядит так:

```
float d3d::Lerp(float a, float b, float t)
{
     return a - (a*t) + (b*t);
}
```

13.6 Пример приложения: ландшафт

Пример приложения для этой главы создает ландшафт на основе содержащихся в файле RAW данных карты высот, текстурирует его и рассчитывает освещение. Помимо этого, с помощью клавиш управления курсором можно перемещаться по ландшафту. Обратите внимание, что в приведенных ниже фрагментах не относящийся к рассматриваемой теме код пропущен; места, где должен находиться пропущенный код отмечены многоточиями. Имейте в виду, что скорость работы приложения зависит от установленного оборудования; если программа работает медленно, попробуйте уменьшить размер ландшафта.

Сперва мы добавляем глобальные переменные, представляющие наш ландшафт, камеру и счетчик частоты кадров:

```
Terrain* TheTerrain = 0;
Camera TheCamera(Camera::LANDOBJECT);

FPSCounter* FPS = 0;
```

Теперь посмотрите на функции каркаса приложения:

```
bool Setup()
{
     D3DXVECTOR3 lightDirection(0.0f, -1.0f, 0.0f);
     TheTerrain = new Terrain(Device, "coastMountain256.raw",
                              256, 256, 10, 1.0f);
     TheTerrain->genTexture();
     TheTerrain->lightTerrain(&directionToLight);

     ...

     return true;
}

void Cleanup()
{
     d3d::Delete<Terrain*>(TheTerrain);
     d3d::Delete<FPSCounter*>(FPS);
```

```
}

bool Display(float timeDelta)
{
     if(Device)
     {
          //
          // Обновление сцены:
          //

          ...[пропущен код для проверки ввода]

          // Ходьба по ландшафту: настраиваем высоту камеры таким
          // образом, чтобы она находилась в 5 единицах над
          // поверхностью той ячейки, где мы находимся.
          D3DXVECTOR3 pos;
          TheCamera.getPosition(&pos);

          float height = TheTerrain->getHeight(pos.x, pos.z);

          pos.y = height + 5.0f;

          TheCamera.setPosition(&pos);

          D3DXMATRIX V;
          TheCamera.getViewMatrix(&V);
          Device->SetTransform(D3DTS_VIEW, &V);

          //
          // Рисование сцены:
          //
          Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                        0xff000000, 1.0f, 0);
          Device->BeginScene();

          D3DXMATRIX I;
          D3DXMatrixIdentity(&I);

          if(TheTerrain)
               TheTerrain->draw(&I, false);

          if(FPS)
               FPS->render(0xffffffff, timeDelta);

          Device->EndScene();
          Device->Present(0, 0, 0, 0);
     }
     return true;
}
```

13.7 Возможные усовершенствования

Рассмотренная реализация класса **Terrain** загружает данные всех вершин ландшафта в один огромный буфер вершин. С точки зрения скорости и масштабируемости было бы хорошо разделить геометрию ландшафта на несколько буферов вершин. Возникает вопрос: «Какой размер буфера лучше выбрать?». Ответ зависит от конфигурации аппаратуры компьютера. Так что экспериментируйте!

Поскольку деление геометрии ландшафта на несколько небольших буферов вершин в основном связано с индексацией в подобной матрице структуре данных и управленим данными, а также не вносит никаких новых концепций, мы не станем подробно обсуждать его. Если коротко, вы делите ландшафт на матрицу элементов, которые мы будем называть «блоки». Каждый блок охватывает прямоугольный фрагмент ландшафта и содержит описание геометрии этого фрагмента (в собственных буферах вершин и индексов). Следовательно, каждый блок отвечает за рисование той части ландшафта, данные которой он содержит.

Альтернативным способом является загрузка всей геометрии ландшафта в один большой экземпляр интерфейса **ID3DXMesh**. Затем для разделения сетки ландшафта на несколько меньших используется функция **D3DXSplitMesh** из библиотеки D3DX. Прототип функции **D3DXSplitMesh** выглядит так:

```
void D3DXSplitMesh(
     const LPD3DXMESH pMeshIn,
     const DWORD *pAdjacencyIn,
     const DWORD MaxSize,
     const DWORD Options,
     DWORD *pMeshesOut,
     LPD3DXBUFFER *ppMeshArrayOut,
     LPD3DXBUFFER *ppAdjacencyArrayOut,
     LPD3DXBUFFER *ppFaceRemapArrayOut,
     LPD3DXBUFFER *ppVertRemapArrayOut
);
```

Эта функция получает исходную сетку и делит ее на несколько меньших. Параметр **pMeshIn** — это указатель на сетку, которую мы хотим разделить, а **pAdjacencyIn** — это указатель на массив данных о смежности для этой сетки. Параметр **MaxSize** используется для указания максимального количества вершин в получаемых в результате разделения фрагментах сетки. Набор флагов **Options** задает параметры создания для формируемых сеток-фрагментов. Через параметр **pMeshesOut** возвращается количество полученных фрагментов, а сами фрагменты возвращаются в массиве буферов **ppMeshArrayOut**. Последние три параметра — необязательные (чтобы игнорировать их укажите **null**) и возвращают массивы данных о смежности и информацию о перемещении граней и вершин для каждой из созданных сеток.

13.8 Итоги

- Мы можем моделировать ландшафты используя сетку с треугольными ячейками, вершинам которой присвоены различные высоты для создания возвышенностей и низин, имитирующих ландшафт.
- Карта высот — это набор данных, содержащий значения высот для каждой вершины ландшафта.
- Мы можем текстурировать ландшафт, используя в качестве текстуры хранящееся на диске изображение, либо создавая текстуру программно.
- Мы освещаем ландшафт путем вычисления коэффициента затенения для каждого квадрата, определяющего насколько светлым или темным должен быть квадрат. Коэффициент затенения зависит от угла под которым свет падает на квадрат.
- Чтобы камера имитировала «ходьбу» по ландшафту нам нужно уметь определять на каком треугольнике сетки ландшафта мы находимся в данный момент. После этого мы формируем два вектора, совпадающих со сторонами этого треугольника. После этого высота определяется путем линейной интерполяции этих векторов с использованием координат по осям X и Z в нормализованной ячейке ландшафта, начало которой совпадает с началом координат, а размеры сторон равны единице.

Глава 14
Системы частиц

Немало природных явлений представляет собой совокупность небольших частиц, ведущих себя одинаковым образом (например, отдельные снежинки снегопада, искры фейерверка и дымовые следы от снарядов). Для моделирования таких явлений используются системы частиц.

Цели

- Изучить атрибуты, которые мы назначаем частицы и то, как представить частицу в Direct3D
- Разработать гибкий базовый класс системы частиц, включающий атрибуты и методы общие для всех систем частиц.
- Создать модели трех систем частиц: снегопада, фейерверка и следа от снаряда.

14.1 Частицы и точечные спрайты

Частицы — это очень маленькие объекты, которые обычно математически моделируются как точки. Из этого следует, что примитивы точек (**D3DPT_POINTLIST** из **D3DPRIMITIVETYPE**) стоят в первых строчках списка кандидатов на отображение частиц. Однако примитивы точек визуализируются как единственная точка. Это не предоставляет нам достаточной гибкости, поскольку могут требоваться частицы разного размера и текстуры частиц могут даже наноситься на карту. даже вся карта может быть тексурирована этими частицами. До Direct3D 8.0 способ обойти накладываемые на примитивы точек ограничения состоял в том, чтобы не использовать их вообще. Вместо этого для отображения частиц программисты использовали *щиты* (*billboard*). Щит — это прямоугольник, который мировая матрица всегда ориентирует так, чтобы лицевой стороной он был обращен к камере.

В Direct3D 8.0 появился специальный точечный примитив, называемый *точечный спрайт* (*point sprite*), который замечательно подходит для реализации систем частиц. В отличие от обычных точечных примитивов точечному спрайту может быть назначена текстура, которая может изменять свой размер. В отличие от щита мы можем описать точечный спрайт с помощью одной точки, что экономит память и время процессора, ведь нам надо обрабатывать одну вершину, а не четыре как у щита (прямоугольника).

14.1.1 Формат структуры

Для описния местоположения и цвета наших частиц мы будем использовать следующий формат вершин:

```
struct Particle
{
     D3DXVECTOR3  _position;
     D3DCOLOR     _color;
     static const DWORD FVF;
};
const DWORD Particle::FVF = D3DFVF_XYZ | D3DFVF_DIFFUSE;
```

В структуре просто хранятся местоположение вершины и ее цвет. В зависимости от требований вашего приложения вы можете хранить также и набор координат текстур. Текстурирование точечных спрайтов мы рассмотрим в следующем разделе.

В структуру **Particle** можно добавить переменную с плавающей точкой, задающую размер частицы. Для этого к описанию настраиваемого формата вершин следует добавить флаг **D3DFVF_PSIZE**. Если частица может иметь собственный размер, мы сможем реализовать множество эффектов, основанных на изменении размеров отдельных частиц. Однако большинство видеокарт не поддерживают такой способ управления размером частиц, поэтому мы не будем обсуждать его. (Чтобы убедиться, что ваша видеокарта поддерживает эту

возможность проверьте флаг **D3DFVFCAPS_PSIZE** в члене **FVFCaps** структуры **D3DCAPS9**.) Вместо этого мы будем управлять размером частиц с помощью режимов визуализации, как показано ниже. Вот пример структуры данных вершины с членом, определяющим размер:

```
struct Particle
{
     D3DXVECTOR3  _position;
     D3DCOLOR     _color;
     float        _size;
     static const DWORD FVF;
};
const DWORD Particle::FVF = D3DFVF_XYZ | D3DFVF_DIFFUSE |
                            D3DFVF_PSIZE;
```

Обратите внимание, что даже если возможность **D3DFVFCAPS_PSIZE** не поддерживается, можно управлять размерами отдельных частиц с помощью вершинных шейдеров. Вершинные шейдеры рассматриваются в четвертой части этой книги.

14.1.2 Режимы визуализации точечных спрайтов

Поведение точечных спрайтов в основном контролируется через режимы визуализации. Сейчас мы рассмотрим эти режимы.

- **D3DRS_POINTSPRITEENABLE** — Логическое значение. Значение по умолчанию — **false**.
    - **True** указывает, что установленная в данный момент текстура накладывается на точечные спрайты целиком.
    - **False** указывает, что на точечный спрайт накладывается только тот тексель текстуры, который задан координатами текстуры точечного спрайта (если координаты текстуры присутствуют в структуре данных вершины точечного спрайта).

```
_device->SetRenderState(D3DRS_POINTSPRITEENABLE, true);
```

- **D3DRS_POINTSCALEENABLE** — Логическое значение. Значение по умолчанию — **false**.
    - **True** указывает, что размер точки интерпретируется в единицах пространства вида. Единицы пространства вида просто ссылаются на трехмерную точку в пространстве камеры. В этом случае размер точечного спрайта масштабируется в зависимости от того, как далеко он находится. В результате, подобно всем другим объектам, частицы расположенные далеко от камеры будут выглядеть меньше, чем частицы, расположенные близко к камере.

- **False** указывает, что размер точки интерпретируется в единицах экранного пространства. Единицы экранного пространства это отображаемые на экране пиксели. Так что если вы укажете **false** и, например, установите размер точечного спрайта равным 3, то он будет занимать на экране область размером 3 × 3 пикселя.

```
_device->SetRenderState(D3DRS_POINTSCALEENABLE, true);
```

- **D3DRS_POINTSIZE** — Используется для задания размера точечных спрайтов. Значение интерпретируется либо как единицы пространства вида, либо как единицы экранного пространства, в зависимости от установленного значения режима **D3DRS_POINTSCALEENABLE**. Приведенный ниже фрагмент кода устанавливает размер точки равным 2.5 единицам:

```
_device->SetRenderState(D3DRS_POINTSIZE, d3d::FtoDw(2.5f));
```

  Функция **d3d::FtoDw** — это вспомогательная функция, добавленная нами в файлы d3dUtility.h/cpp, которая выполняет приведение типа **float** к типу **DWORD**. Нам приходится выполнять эту операцию потому что функция **IDirect3DDevice9::SetRenderState** ожидает значения типа **DWORD** а не **float**.

```
DWORD d3d::FtoDw(float f)
{
     return *((DWORD*)&f);
}
```

- **D3DRS_POINTSIZE_MIN** — Задает минимальный размер точечного спрайта. Приведенный ниже пример устанавливает минимальный размер равным 0.2:

```
_device->SetRenderState(D3DRS_POINTSIZE_MIN,
                        d3d::FtoDw(0.2f));
```

- **D3DRS_POINTSIZE_MAX** — Задает максимальный размер точечного спрайта. Приведенный ниже пример устанавливает максимальный размер равным 5.0:

```
_device->SetRenderState(D3DRS_POINTSIZE_MAX,
                        d3d::FtoDw(5.0f));
```

- **D3DRS_POINTSCALE_A**, **D3DRS_POINTSCALE_B**, **D3DRS_POINTSCALE_C** — Эти три константы позволяют управлять тем, как будет меняться размер точечного спрайта при изменении расстояния от него до камеры.

Для вычисления итогового размера точечного спрайта на основании расстояния до камеры и рассматриваемых констант Direct3D использует следующую формулу:

$$FinalSize = ViewportHeight \cdot Size \cdot \sqrt{\frac{1}{A + B \cdot D + C \cdot D^2}}$$

где:

- *FinalSize* — Итоговый размер точечного спрайта после вычислений расстояния.
- *ViewportHeight* — Высота порта просмотра.
- *Size* — Исходный размер, определяемый установленным значением режима визуализации **D3DRS_POINTSIZE**.
- *A*, *B*, *C* — Значения, заданные режимами визуализации **D3DRS_POINTSCALE_A**, **D3DRS_POINTSCALE_B** и **D3DRS_POINTSCALE_C** соответственно.
- *D* — Расстояние от точечного спрайта до камеры в пространстве вида. Поскольку в пространстве вида камера находится в начале координат, это расстояние вычисляется по формуле

  $$D = \sqrt{x^2 + y^2 + z^2}$$

  где (x, y, z) — это координаты местоположения точечного спрайта в пространстве вида.

Приведенный ниже пример устанавливает значения констант таким образом, чтобы с увеличением расстояния размер точечных спрайтов уменьшался:

```
_device->SetRenderState(D3DRS_POINTSCALE_A, d3d::FtoDw(0.0f));
_device->SetRenderState(D3DRS_POINTSCALE_B, d3d::FtoDw(0.0f));
_device->SetRenderState(D3DRS_POINTSCALE_C, d3d::FtoDw(1.0f));
```

14.1.3 Частицы и их атрибуты

У частиц может быть множество других атрибутов, помимо местоположения и цвета; например, у каждой частицы может быть своя скорость. Однако, эти дополнительные атрибуты не нужны для визуализации частицы. Соответственно мы храним данные для визуализации частицы и дополнительные атрибуты в разных структурах. Создавая, уничтожая и обновляя частицы мы работаем с их атрибутами; затем, когда все готово к визуализации, мы копируем местоположение и цвет частицы в структуру **Particle**.

Атрибуты частицы зависят от того, какую именно модель частиц мы создаем. Однако, мы можем выделить несколько общих атрибутов, которые перечислены в приведенном ниже примере структуры данных. Большинству систем частиц не потребуются все эти атрибуты, а для некоторых систем надо будет добавить дополнительные атрибуты отсутствующие в списке.

```
struct Attribute
{
     D3DXVECTOR3 _position;
     D3DXVECTOR3 _velocity;
     D3DXVECTOR3 _acceleration;
     float       _lifeTime;
     float       _age;
     D3DXCOLOR   _color;
     D3DXCOLOR   _colorFade;
     bool        _isAlive;
};
```

- **_position** — Местоположение частицы в мировом пространстве.
- **_velocity** — Скорость частицы, обычно измеряемая в условных единицах в секунду.
- **_acceleration** — Ускорение частицы, обычно измеряемое в условных единицах за секунду.
- **_lifeTime** — Сколько времени должно пройти до гибели частицы. Например, мы можем указать, что частицы образующие лазерный луч пропадают через указанный период времени.
- **_age** — Текущий возраст частицы.
- **_color** — Цвет частицы.
- **_colorFade** — Как цвет частицы меняется с течением времени.
- **_isAlive** — **True** если частица жива, **false** если она погибла.

14.2 Компоненты системы частиц

Системой частиц называется набор частиц и код, отвечающий за управление этими частицами и их отображение. Система частиц отслеживает глобальные свойства, влияющие на все частицы, такие как размер частиц, место из которого появляются частицы, накладываемая на частицы текстура и т.д. С точки зрения функциональности система частиц отвечает за обновление, отображение, уничнтожение и создание частиц.

Хотя различные системы частиц ведут себя по-разному, мы можем выполнить обобщение и выделить некоторые базовые свойства, которые используются всеми системами частиц. Мы поместим эти общие свойства в абстрактный базовый класс **PSystem**, который будет родителем для классов конкретных систем частиц. Давайте теперь взглянем на класс **PSystem**:

```
class PSystem
{
public:
     PSystem();
     virtual ~PSystem();
```

```
     virtual bool init(IDirect3DDevice9* device, char* texFileName);
     virtual void reset();
     virtual void resetParticle(Attribute* attribute) = 0;
     virtual void addParticle();
     virtual void update(float timeDelta) = 0;

     virtual void preRender();
     virtual void render();
     virtual void postRender();

     bool isEmpty();
     bool isDead();
protected:
     virtual void removeDeadParticles();

protected:
     IDirect3DDevice9*       _device;
     D3DXVECTOR3             _origin;
     d3d::BoundingBox        _boundingBox;
     float                   _emitRate;
     float                   _size;
     IDirect3DTexture9*      _tex;
     IDirect3DVertexBuffer9* _vb;
     std::list<Attribute>    _particles;
     int                     _maxParticles;

     DWORD _vbSize;
     DWORD _vbOffset;
     DWORD _vbBatchSize;
};
```

Начнем с членов данных:

- **_origin** — Базовая точка системы. Это то место, откуда появляются частицы системы.
- **_boundingBox** — Ограничивающий параллелепипед используется в тех системах частиц, где надо ограничить объем пространства в котором могут находиться частицы. Например, мы хотим, чтобы снег падал только в пространстве, окружающем высокую вершину горы; для этого следует создать ограничивающий параллелепипед, охватывающий желаемый объем и частицы, вышедшие за границы этого объема будут уничтожаться.
- **_emitRate** — Частота добавления новых частиц к системе. Обычно измеряется в частицах в секунду.
- **_size** — Размер всех частиц системы.
- **_particles** — Список, содержащий атрибуты частиц системы. Мы работаем с этим списком при создании, уничножении и обновлении частиц. Когда мы готовы к рисованию частиц, мы копируем часть узлов списка в буфер вершин и рисуем частицы. Затем мы копируем следующий блок и рисуем частицы. Эти действия повторяются до тех пор, пока не

будут нарисованы все частицы. Это крайне упрощенное описание; подробно процесс рисования будет рассмотрен в разделе 14.2.1.

- **_maxParticles** — Максимальное количество частиц, которое может быть в системе одновременно. Если, к примеру, частицы создаются быстрее чем удаляются, может получиться, что у вас будет огромное количество частиц, что приведет к неработоспособности программы. Данный член позволяет избежать такого развития событий.
- **_vbSize** — Количество частиц, которое может быть помещено в буфер вершин одновременно. Это значение не зависит от количества частиц в конкретной системе частиц.

ПРИМЕЧАНИЕ	Члены данных **_vbOffset** и **_vbBatchSize** используются при визуализации системы частиц. Мы отложим их обсуждение до раздела 14.2.1.

Методы класса:

- **PSystem**/**~PSystem** — Конструктор инициализирует значения по умолчанию, а деструктор освобождает интерфейсы устройства (буфер вершин, текстуры).
- **init** — Этот метод выполняет зависящую от устройства Direct3D инициализацию, такую как создание буфера вершин для хранения точечных спрайтов и создание текстуры. При создании буфера вершин указываются несколько флагов о которых мы говорили, но до настоящего момента еще ни разу не использовали:

```
hr = device->CreateVertexBuffer(
      _vbSize * sizeof(Particle),
      D3DUSAGE_DYNAMIC | D3DUSAGE_POINTS | D3DUSAGE_WRITEONLY,
      Particle::FVF,
      D3DPOOL_DEFAULT,
      &_vb,
      0);
```

  - Обратите внимание, что мы используем динамический буфер вершин. Это вызвано тем, что мы должны в каждом кадре обновлять данные частиц, что требует доступа к памяти буфера вершин. Вспомните, что доступ к статическому буферу вершин осуществляется очень медленно; поэтому мы используем динамический буфер вершин.
  - Взгляните на используемый флаг **D3DUSAGE_POINTS**, который сообщает, что в буфере вершин будут храниться точечные спрайты.
  - Обратите внимание, что размер буфера вершин задан переменной **_vbSize** и не имеет ничего общего с количеством частиц в системе. То есть, значение **_vbSize** очень редко равно количеству частиц в системе. Это вызвано тем, что мы

визуализируем систему частиц по частям, а не всю сразу. Процесс визуализации мы исследуем в разделе 14.2.1.

- Мы используем пул памяти по умолчанию, а не обычный управляемый пул памяти потому что динамический буфер вершин не может быть размещен в управляемом пуле памяти.

- **reset** — Этот метод сбрасывает значения атрибутов у каждой частицы системы:

```
void PSystem::reset()
{
     std::list<Attribute>::iterator i;
     for(i = _particles.begin(); i != _particles.end(); i++)
     {
          resetParticle(&(*i));
     }
}
```

- **resetParticle** — Этот метод сбрасывает значения атрибутов одной частицы. То, как именно должен выполняться сброс атрибутов, зависит от параметров конкретной системы частиц. Следовательно, мы делаем этот метод абстрактным и он должен быть реализован в производном классе.

- **addParticle** — Этот метод добавляет частицу к системе. Он использует метод **resetParticle** для инициализации частицы перед ее добавлением к списку:

```
void PSystem::addParticle()
{
     Attribute attribute;

     resetParticle(&attribute);

     _particles.push_back(attribute);
}
```

- **update** — Этот метод обновляет данные всех частиц системы. Поскольку реализация такого метода зависит от спецификаций конкретной системы частиц, мы объявляем этот метод абстрактным и должны реализовать его в производном классе.

- **render** — Данный метод отображает все частицы системы. Его реализация достаточно сложна и мы отложим ее обсуждение до раздела 14.2.1.

- **preRender** — Применяется для установки начальных режимов визуализации, которые должны быть заданы перед визуализацией. Поскольку они могут меняться в зависимости от конкретной системы частиц, мы делаем этот метод виртуальным. Предлагаемая по умолчанию реализация выглядит так:

```
void PSystem::preRender()
{
     _device->SetRenderState(D3DRS_LIGHTING, false);
     _device->SetRenderState(D3DRS_POINTSPRITEENABLE, true);
     _device->SetRenderState(D3DRS_POINTSCALEENABLE, true);
     _device->SetRenderState(D3DRS_POINTSIZE,
                                    d3d::FtoDw(_size));
     _device->SetRenderState(D3DRS_POINTSIZE_MIN,
                                    d3d::FtoDw(0.0f));

     // Управление изменением размера частицы
     // в зависимости от расстояния до нее
     _device->SetRenderState(D3DRS_POINTSCALE_A,
                                    d3d::FtoDw(0.0f));
     _device->SetRenderState(D3DRS_POINTSCALE_B,
                                    d3d::FtoDw(0.0f));
     _device->SetRenderState(D3DRS_POINTSCALE_C,
                                    d3d::FtoDw(1.0f));

     // Для текстуры используется альфа-смешивание
     _device->SetTextureStageState(0, D3DTSS_ALPHAARG1,
                                          D3DTA_TEXTURE);
     _device->SetTextureStageState(0, D3DTSS_ALPHAOP,
                                      D3DTOP_SELECTARG1);
     _device->SetRenderState(D3DRS_ALPHABLENDENABLE, true);
     _device->SetRenderState(D3DRS_SRCBLEND,
                                      D3DBLEND_SRCALPHA);
     _device->SetRenderState(D3DRS_DESTBLEND,
                                    D3DBLEND_INVSRCALPHA);
}
```

Обратите внимание, что мы разрешили альфа-смешивание, чтобы альфа-канал установленной в данный момент текстуры определял прозрачность ее участков. Прозрачность используется для множества эффектов; один из них — реализация частиц непрямоугольной формы. Предположим, вам нужны круглые, похожие на хлопья снега, частицы. В этом случае мы используем текстуру в виде белого квадрата с альфа-каналом в виде черного квадрата с белым кругом. В результате на экране будет отображен похожий на снежинку белый круг, а не квадратная текстура.

- **`postRender`** — Используется для восстановления режимов визуализации, которые изменила данная система частиц. Поскольку режимы меняются в зависимости от конкретной системы частиц, этот метод виртуальный. По умолчанию используется следующая реализация:

```
void PSystem::postRender()
{
     _device->SetRenderState(D3DRS_LIGHTING,          true);
     _device->SetRenderState(D3DRS_POINTSPRITEENABLE, false);
     _device->SetRenderState(D3DRS_POINTSCALEENABLE,  false);
     _device->SetRenderState(D3DRS_ALPHABLENDENABLE,  false);
}
```

- **isEmpty** — Метод возвращает **true**, если в системе нет ни одной частицы и **false** в ином случае.
- **isDead** — Метод возвращает **true** если все частицы в системе мертвы и **false**, если хотя бы одна частица жива. Обратите внимание что если все частицы мертвы, это не значит, что система частиц пуста. В пустой системе нет ни мертвых ни живых частиц. Если система мертвая, это значит, что в ней есть частицы, но все они помечены как мертвые.
- **removeDeadParticles** — Метод перебирает элементы списка атрибутов **_particle** и удаляет из него все частицы, которые отмечены как мертвые:

```
void PSystem::removeDeadParticles()
{
     std::list::iterator i;
     i = _particles.begin();
     while(i != _particles.end())
     {
          if( i->_isAlive == false )
          {
               // стирание возвращает номер следующего
               // элемента, поэтому самостоятельно
               // увеличивать счетчик не надо
               i = _particles.erase(i);
          }
          else
          {
               i++; // следующий элемент списка
          }
     }
}
```

ПРИМЕЧАНИЕ	Этот метод обычно вызывается из метода обновления данных частиц производного класса для удаления тех частиц, которые были уничтожены (помечены как мертвые). Однако для некоторых систем частиц может оказаться предпочтительнее повторно использовать мертвые частицы, а не удалять их. В этом случае вместо добавления в список новых частиц и удаления из него старых в моменты их рождения и смерти, мы просто сбрасываем данные мертвой частицы и воскрешаем ее. Этот подход будет продемонстрирован в реализации снегопада, рассмариваемой в разделе 14.3.

14.2.1 Рисование системы частиц

Поскольку система частиц является динамической, нам надо обновлять данные частиц в каждом кадре. Первый приходящий на ум, но неэффективный метод визуализации системы частиц заключается в следующем:

1. Создать буфер вершин, размер которого достаточен для хранения всех находящихся в кадре частиц.

   В каждом кадре:

   A. Обновить данные всех частиц.

   B. Скопировать все живые частицы в буфер вершин.

   C. Нарисовать содержимое буфера вершин.

Этот подход работает, но неэффективен. Во-первых, для того чтобы хранить все частицы системы, буфер вершин должен быть очень большим. Что еще более важно, в момент копирования данных частиц из списка в буфер вершин (этап B) видеокарта будет простаивать. Предположим, что в нашей системе 10 000 частиц; для начала нам понадобится буфер вершин, в котором могут храниться данные 10 000 частиц, а он займет очень много памяти. Помимо этого, видеокарта будет ждать и ничего не делать пока мы не скопируем данные всех 10 000 частиц из списка в буфер вершин и не вызовем метод **`DrawPrimitive`**. Этот сценарий хороший пример того, как центральный процессор и видеокарта *не* работают сообща.

Гораздо лучший подход (который используется в примере Point Sprite из SDK) заключается в следующем:

- Создайте буфер вершин большого размера (скажем, такой, в котором может храниться 2000 частиц). Затем разделите буфер на сегменты; в нашем примере размер сегмента будет равен 500 частицам.

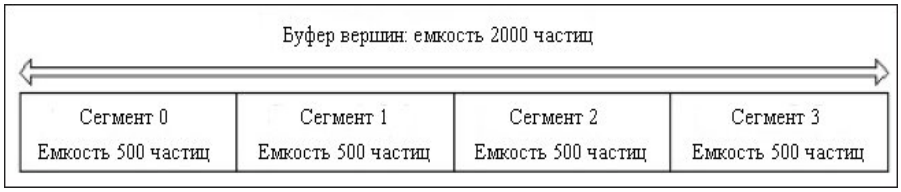

Рис. 14.1. *Разделенный на сегменты буфер вершин*

Теперь создайте глобальную переменную $i = 0$ для отслеживания того, с каким сегментом мы работаем.

Для каждого кадра:

A. Обновите все частицы.

B. Пока не будут визуализированы все живые частицы:

   1. Если буфер вершин не заполнен, то:

      a. Блокируем сегмент *i* с флагом **`D3DLOCK_NOOVERWRITE`**.

      b. Копируем 500 частиц в сегмент *i*.

   2. Если буфер вершин заполнен, то:

      a. Возвращаемся к началу буфера вершин: $i = 0$.

      b. Блокируем сегмент *i* с флагом **`D3DLOCK_DISCARD`**.

c. Копируем 500 частиц в сегмент *i*.

3. Визуализируем сегмент *i*.
4. Переходим к следующему сегменту: *i*++

ПРИМЕЧАНИЕ Это очень упрощенное описание, но оно иллюстрирует идею. Предполагается, что у нас всегда есть 500 частиц, чтобы заполнить целый сегмент, хотя в действительности это условие не выполняется потому что мы постоянно создаем и уничтожаем частицы и их количество меняется от кадра к кадру. Предположим, что у нас осталось только 200 частиц для копирования и визуализации в текущем кадре. Поскольку 200 частиц недостаточно для заполнения буфера вершин, в коде мы обрабатываем такой сценарий как особый случай. Данный сценарий может иметь место только в том случае, когда в текущем кадре заполняется последний сегмент. Если сегмент не последний, это значит что у нас есть по крайней мере 500 частиц для перехода к следующему сегменту.

ПРИМЕЧАНИЕ Вспомните, что наш буфер вершин динамический, и поэтому мы можем пользоваться преимуществами, предоставляемыми флагами динамической блокировки **`D3DLOCK_NOOVERWRITE`** и **`D3DLOCK_DISCARD`**. Эти флаги позволяют блокировать часть буфера вершин, которая не визуализируется, в то время как остальные части буфера вершин будут продолжать визуализироваться. Например, предположим, что мы визуализируем сегмент 0; используя флаг **`D3DLOCK_NOOVERWRITE`** мы можем заблокировать и заполнить сегмент 1 в то время, когда визуализируется сегмент 0. Это позволяет предотвратить простои визуализации, которые возникали бы в ином случае.

Данный подход более эффективен. Во-первых, мы сокращаем размер необходимого нам буфера вершин. Во-вторых, теперь центральный процессор и видеокарта работают в унисон; то есть мы копируем небольшую партию частиц в буфер вершин (работа центрального процессора), а затем мы визуализируем эту партию частиц (работа видеокарты). Затем мы копируем в буфер вершин следующую партию частиц и рисуем ее. Это продолжается до тех пор, пока не будут визуализированы все частицы. Как видите, видеокарта больше не простаивает, ожидая пока не будет заполнен весь буфер вершин.

Теперь мы обратим наше внимание на реализацию этой схемы визуализации. Чтобы облегчить визуализацию системы частиц с помощью данной схемы мы будем использовать следующие члены данных класса **`PSystem`**:

- **`_vbSize`** — Количество частиц, которые одновременно могут храниться в нашем буфере вершин. Это значение не зависит от количества частиц в конкретной системе частиц.
- **`_vbOffset`** — Переменная хранит смещение в буфере вершин (измеряемое в частицах, а не в байтах), начиная с которого мы должны выполнять копирование очередной партии частиц. Например, если первая

партия частиц заняла элементы буфера вершин с номерами от 0 до 499, то копирование следующей партии должно начинаться со смещения 500.

- **`_vbBatchSize`** — Количество частиц в партии.

Теперь мы представим вам код метода визуализации:

```
void PSystem::render()
{
     if(!_particles.empty())
     {
          // Установка режимов визуализации
          preRender();
          _device->SetTexture(0, _tex);
          _device->SetFVF(Particle::FVF);
          _device->SetStreamSource(0, _vb, 0, sizeof(Particle));

          // Если мы достигли конца буфера вершин,
          // возвращаемся к его началу
          if(_vbOffset >= _vbSize)
               _vbOffset = 0;

          Particle* v = 0;

          _vb->Lock(
                _vbOffset * sizeof(Particle),
                _vbBatchSize * sizeof(Particle),
                (void**)&v,
                _vbOffset ? D3DLOCK_NOOVERWRITE : D3DLOCK_DISCARD);

          DWORD numParticlesInBatch = 0;

          //
          // Пока все частицы не будут визуализированы
          //
          std::list<Attribute>::iterator i;
          for(i = _particles.begin(); i != _particles.end(); i++)
          {
               if(i->_isAlive)
               {
                    //
                    // Копируем партию живых частиц в
                    // очередной сегмент буфера вершин
                    //
                    v->_position = i->_position;
                    v->_color = (D3DCOLOR)i->_color;
                    v++; // следующий элемент;

                    numParticlesInBatch++; // увеличиваем счетчик
                                           // партий

                    // партия полная?
                    if(numParticlesInBatch == _vbBatchSize)
                    {
                         //
```

```
                    // Рисуем последнюю партию частиц, которая
                    // была скопирована в буфер вершин.
                    //
                    _vb->Unlock();
                    _device->DrawPrimitive(
                                D3DPT_POINTLIST,
                                _vbOffset,
                                _vbBatchSize);
                    //
                    // Пока партия рисуется, начинаем заполнять
                    // следующую партию частиц.
                    //
                    // Увеличиваем смещение к началу
                    // следующей партии
                    _vbOffset += _vbBatchSize;
                    // Проверяем не вышли ли мы за пределы
                    // буфера вершин. Если да, то возвращаемся
                    // к началу буфера.
                    if(_vbOffset >= _vbSize)
                         _vbOffset = 0;
                    _vb->Lock(
                          _vbOffset * sizeof(Particle),
                          _vbBatchSize * sizeof(Particle),
                          (void**)&v,
                          _vbOffset ? D3DLOCK_NOOVERWRITE :
                                      D3DLOCK_DISCARD);
                    numParticlesInBatch = 0; // обнуляем коли-
                                             // чество частиц в
                                             // партии
               } //конец инструкции if
          } //конец инструкции if
     } //конец инструкции for
     _vb->Unlock();
     // Возможно, ПОСЛЕДНЯЯ партия частиц начала заполняться,
     // но не была визуализирована, потому что условие
     // (numParticlesInBatch == _vbBatchSize) не было выполнено.
     // Сейчас мы нарисуем эту последнюю частично заполненную
     // партию частиц
     if( numParticlesInBatch )
     {
          _device->DrawPrimitive(
                         D3DPT_POINTLIST,
                         _vbOffset,
                         numParticlesInBatch);
     }
     // Следующий блок
     _vbOffset += _vbBatchSize;
     postRender();
  } //конец инструкции if
} // конец метода render()
```

14.2.2 Хаотичность

В системах частиц есть своего рода хаотичность. Например, моделируя снегопад мы не хотим, чтобы все снежинки падали абсолютно одинаково. Нам нужно чтобы они падали похожим образом, а не абсолютно одинаково. Чтобы облегчить реализацию хаотичности, необходимую для систем частиц, мы добавляем в файлы d3dUtility.h/cpp две функции.

Первая функция возвращает случайное число с плавающей точкой, находящееся в диапазоне [lowBound, highBound]:

```
float d3d::GetRandomFloat(float lowBound, float highBound)
{
     if(lowBound >= highBound) // неправильные параметры
          return lowBound;

     // Получаем случайное число в диапазоне [0, 1]
     float f = (rand() % 10000) * 0.0001f;

     // Возвращаем число из диапазона [lowBound, highBound]
     return (f * (highBound - lowBound)) + lowBound;
}
```

Следующая функция возвращает случайный вектор в параллелепипеде, заданном двумя углами **min** и **max**.

```
void d3d::GetRandomVector(
                  D3DXVECTOR3* out,
                  D3DXVECTOR3* min,
                  D3DXVECTOR3* max)
{
     out->x = GetRandomFloat(min->x, max->x);
     out->y = GetRandomFloat(min->y, max->y);
     out->z = GetRandomFloat(min->z, max->z);
}
```

ПРИМЕЧАНИЕ Не забывайте об инициализации генератора случайных чисел с помощью функции **srand()**.

14.3 Примеры систем частиц: снег, фейерверк, след снаряда

Давайте теперь на основе класса **PSystem** реализуем несколько конкретных систем частиц. Эти системы будут достаточно просты, поскольку разработаны с целью иллюстрации, и не будут в полной мере использовать преимущества гибкости, предоставляемой классом **PSystem**. Мы реализуем системы снегопада, фейерверка и следа снаряда. Имена этих систем явно указывают на то, какие явления они моделируют. Система снегопада моделирует падение снежинок. Система фейерверка моделирует взрыв, похожий на залп салюта. Система следа снаряда при нажатии клавиши запускает частицы из точки расположения камеры

в том направлении, куда камера направлена, создавая впечатление, что мы запустили ракету. Она может использоваться как основа для имитации стрельбы из оружия в игре.

ПРИМЕЧАНИЕ Как обычно, полный код проектов иллюстрирующих эти системы частиц находится в сопроводительных файлах к этой главе.

14.3.1 Пример приложения: снег

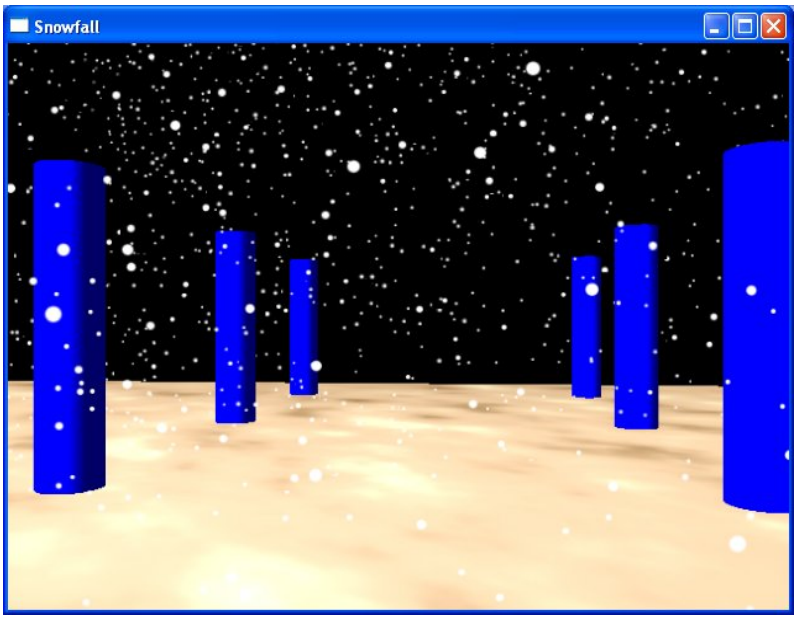

Рис. 14.2. *Окно программы Snow*

Определение класса системы частиц **`Snow`** выглядит следующим образом:

```
class Snow : public Psystem
{
public:
     Snow(d3d::BoundingBox* boundingBox, int numParticles);
     void resetParticle(Attribute* attribute);
     void update(float timeDelta);
};
```

ПРИМЕЧАНИЕ Обратите внимание насколько прост интерфейс класса системы частиц **`Snow`**. Это объясняется тем, что большую часть работы выполняет родительский класс. Фактически, все три системы частиц, которые мы рассматриваем в этом разделе, имеют достаточно простые интерфейсы, которые относительно легко реализовать.

Конструктор получает указатель на структуру данных ограничивающего параллелепипеда и количество частиц, которое должно быть в системе. Ограничивающий параллелепипед определяет объем пространства в котором будет падать снег. Если снежинки выходят за границы этого объема, они воскрешаются с новыми координатами. Таким образом система поддерживает постоянное количество активных частиц. Реализация конструктора выглядит следующим образом:

```
Snow::Snow(d3d::BoundingBox* boundingBox, int numParticles)
{
     _boundingBox = *boundingBox;
     _size = 0.8f;
     _vbSize = 2048;
     _vbOffset = 0;
     _vbBatchSize = 512;

     for(int i = 0; i < numParticles; i++)
          addParticle();
}
```

Обратите внимание, что мы задаем размер буфера вершин, размер партии частиц и начальное смещение.

Метод **`resetParticle`** создает снежинку внутри ограничивающего параллелепипеда со случайными значениями координат X и Z, а значение координаты Y делает равным координате верха ограничивающего объема. Затем вектор скорости снежинки устанавливается таким образом, чтобы она падала вниз и при этом слегка смещалась влево. Помимо этого, снежинка окрашивается в белый цвет:

```
void Snow::resetParticle(Attribute* attribute)
{
     attribute->_isAlive = true;

     // Получить случайные значения координат X и Z снежинки
     d3d::GetRandomVector(
               &attribute->_position,
               &_boundingBox._min,
               &_boundingBox._max);

     // Для высоты (координаты Y) случайное значение не нужно.
     // Падение снежинок всегда начинается с верха
     // ограничивающего параллелепипеда
     attribute->_position.y = _boundingBox._max.y;

     // Снежинка падает вниз и слегка смещается влево
     attribute->_velocity.x=d3d::GetRandomFloat(0.0f, 1.0f) * -3.0f;
     attribute->_velocity.y=d3d::GetRandomFloat(0.0f, 1.0f) * -10.0f;
     attribute->_velocity.z = 0.0f;

     // Все снежинки белые
     attribute->_color = d3d::WHITE;
}
```

Метод **update** обновляет местоположение частиц, а затем проверяет не вышли ли какие-нибудь частицы за пределы ограничивающего объема системы. Если частица вышла за пределы ограничивающего параллелепипеда, она заново инициализируется и снова падает с самого верха.

```
void Snow::update(float timeDelta)
{
     std::list<Attribute>::iterator i;
     for(i = _particles.begin(); i != _particles.end(); i++)
     {
          i->_position += i->_velocity * timeDelta;

          // Точка вне ограничивающего объема?
          if(_boundingBox.isPointInside(i->_position) == false)
          {
               // Вышедшие за пределы объема частицы не
               // уничтожаются, а снова используются и
               // воскрешаются с новыми координатами
               resetParticle(&(*i));
          }
     }
}
```

14.3.2 Пример приложения: фейерверк

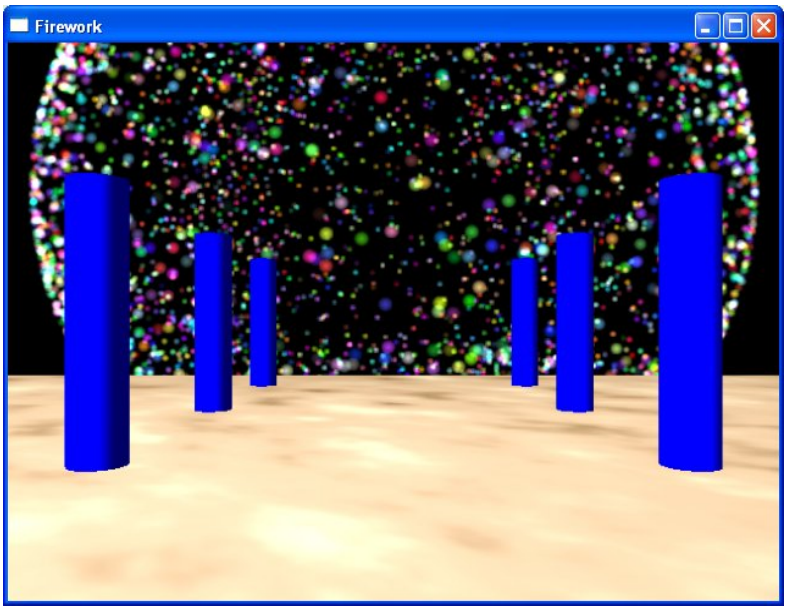

Рис. 14.3. *Окно программы Firework*

Определение класса системы **Firework** выглядит следующим образом:

```
class Firework : public Psystem
{
public:
     Firework(D3DXVECTOR3* origin, int numParticles);
     void resetParticle(Attribute* attribute);
     void update(float timeDelta);
     void preRender();
     void postRender();
};
```

Конструктор получает указатель на базовую точку системы и количество частиц в системе. В данном случае базовая точка — это точка в которой взрывается фейерверк.

Метод **resetParticle** инициализирует частицу, помещая ее в базовую точку системы и назначает ей случайный вектор скорости в сфере. Каждой частице системы **Firework** назначается выбираемый случайным образом цвет. Помимо этого, мы указываем, что время жизни частицы равно двум секундам.

```
void Firework::resetParticle(Attribute* attribute)
{
     attribute->_isAlive = true;
     attribute->_position = _origin;

     D3DXVECTOR3 min = D3DXVECTOR3(-1.0f, -1.0f, -1.0f);
     D3DXVECTOR3 max = D3DXVECTOR3( 1.0f, 1.0f, 1.0f);

     d3d::GetRandomVector(
            &attribute->_velocity,
            &min,
            &max);

     // Нормализация для сферы
     D3DXVec3Normalize(
            &attribute->_velocity,
            &attribute->_velocity);

     attribute->_velocity *= 100.0f;

     attribute->_color = D3DXCOLOR(
            d3d::GetRandomFloat(0.0f, 1.0f),
            d3d::GetRandomFloat(0.0f, 1.0f),
            d3d::GetRandomFloat(0.0f, 1.0f),
            1.0f);

     attribute->_age = 0.0f;
     attribute->_lifeTime = 2.0f; // время жизни - 2 секунды
}
```

Метод **update** обновляет местоположение каждой частицы и уничтожает те из ник, которые просуществовали больше заданного времени. Обратите внимание, что система не удаляет мертвые частицы. Благодаря этому, если мы захотим создать новый фейерверк, нам надо будет просто выполнить инициализацию

мертвой системы фейерверка. Такой подход позволяет избежать частого создания и уничтожения частиц.

```
void Firework::update(float timeDelta)
{
     std::list<Attribute>::iterator i;

     for(i = _particles.begin(); i != _particles.end(); i++)
     {
          // Обновляем только живые частицы
          if(i->_isAlive)
          {
               i->_position += i->_velocity * timeDelta;

               i->_age += timeDelta;

               if(i->_age > i->_lifeTime) // убиваем
                    i->_isAlive = false;
          }
     }
}
```

Система фейерверка при визуализации использует собственные коэффициенты смешивания. Кроме того, она еще и запрещает запись в буфер глубины. Мы можем легко изменить коэффициенты смешивания и запретить запись в буфер глубины путем переопределения методов **`PSystem::preRender`** и **`PSystem::postRender`**. Вот переопределенная реализация:

```
void Firework::preRender()
{
     PSystem::preRender();

     _device->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE);
     _device->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);

     // Не записываем частицы в z-буфер
     _device->SetRenderState(D3DRS_ZWRITEENABLE, false);
}

void Firework::postRender()
{
     PSystem::postRender();

     _device->SetRenderState(D3DRS_ZWRITEENABLE, true);
}
```

Обратите внимание, что оба метода вызывают одноименные методы родительского класса. Это позволяет использовать всю предлагаемую родительским классом функциональность, оставив в классе **`Firework`** только минимум кода, требуемый для данной конкретной системы частиц.

14.3.3 Пример приложения: след снаряда

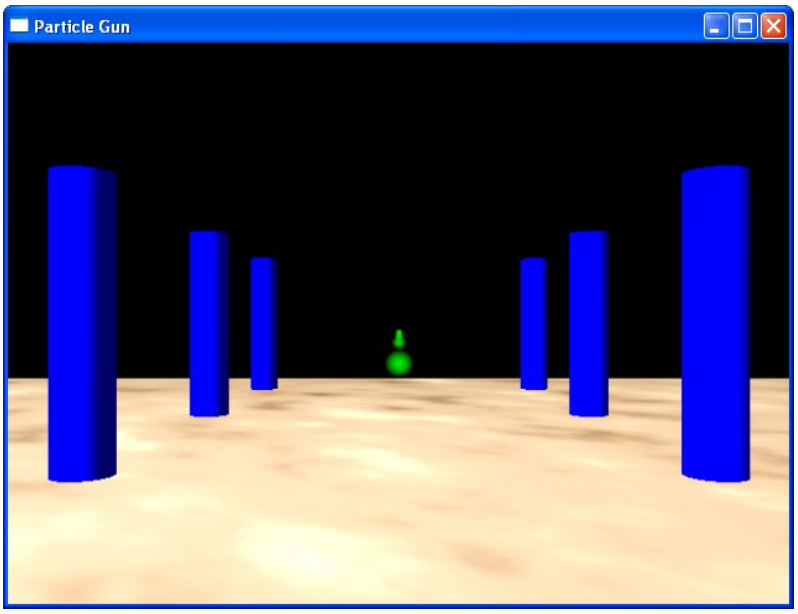

Рис. 14.4. *Окно программы Laser (Particle Gun)*

Определение класса системы **`ParticleGun`** выглядит следующим образом:

```
class ParticleGun : public Psystem
{
public:
     ParticleGun(Camera* camera);
     void resetParticle(Attribute* attribute);
     void update(float timeDelta);

private:
     Camera* _camera;
};
```

Конструктор получает указатель на камеру. Это объясняется тем, что данной системе для создания частиц надо знать местоположение камеры и направление, в котором она смотрит.

Метод **`resetParticle`** устанавливает координаты частицы равными текущим координатам камеры и задает вектор скорости частицы равным умноженному на сто вектору взгляда камеры. В результате «снаряд» будет выстрелен в направлении взгляда. Изображающей снаряд частице мы назначаем зеленый цвет.

```
void ParticleGun::resetParticle(Attribute* attribute)
{
     attribute->_isAlive = true;
```

```
     D3DXVECTOR3 cameraPos;
     _camera->getPosition(&cameraPos);

     D3DXVECTOR3 cameraDir;
     _camera->getLook(&cameraDir);

     // Получаем местоположение камеры
     attribute->_position = cameraPos;
     attribute->_position.y -= 1.0f; // смещаем позицию вниз, чтобы
                                     // казалось, что мы держим
                                     // оружие в руках

     // Отправляем частицу в том направлении, куда смотрит камера
     attribute->_velocity = cameraDir * 100.0f;

     // Назначаем зеленый цвет
     attribute->_color = D3DXCOLOR(0.0f, 1.0f, 0.0f, 1.0f);

     attribute->_age = 0.0f;
     attribute->_lifeTime = 1.0f; // время жизни - 1 секунда
}
```

Метод **`update`** обновляет местоположение частиц и уничтожает частицы, которые прожили больше заданного времени. В самом конце функции мы просматриваем список частиц и удаляем из него все мертвые частицы.

```
void ParticleGun::update(float timeDelta)
{
     std::list<Attribute>::iterator i;

     for(i = _particles.begin(); i != _particles.end(); i++)
     {
          i->_position += i->_velocity * timeDelta;

          i->_age += timeDelta;

          if(i->_age > i->_lifeTime) // убиваем
               i->_isAlive = false;
     }
     removeDeadParticles();
}
```

14.4 Итоги

- Точечные спрайты — это удобный и гибкий способ отображения частиц. Они могут менять размер и на них можно накладывать текстуры. Кроме того, для описания точечного спрайта требуется только одна вершина.
- Система частиц управляет набором частиц и отвечает за создание, уничтожение, обновление и отображение частиц.
- Вот несколько явлений, которые вы можете моделировать с помощью систем частиц: дым, след ракеты, фонтан, огонь, лучи света, взрывы и дождь.

Глава 15
Выбор объектов

Предположим, что пользователь щелкает по точке экрана **s** = (*x*, *y*). На рис. 15.1 видно, что в этом случае пользователь *выбрал* чайник. Однако приложение не может немедленно определить, что при щелчке по точке **s** выбран именно чайник. Существует техника, позволяющая определить, какой именно из объектов сцены выбран. Эта техника называется *выбором объектов* (*picking*).

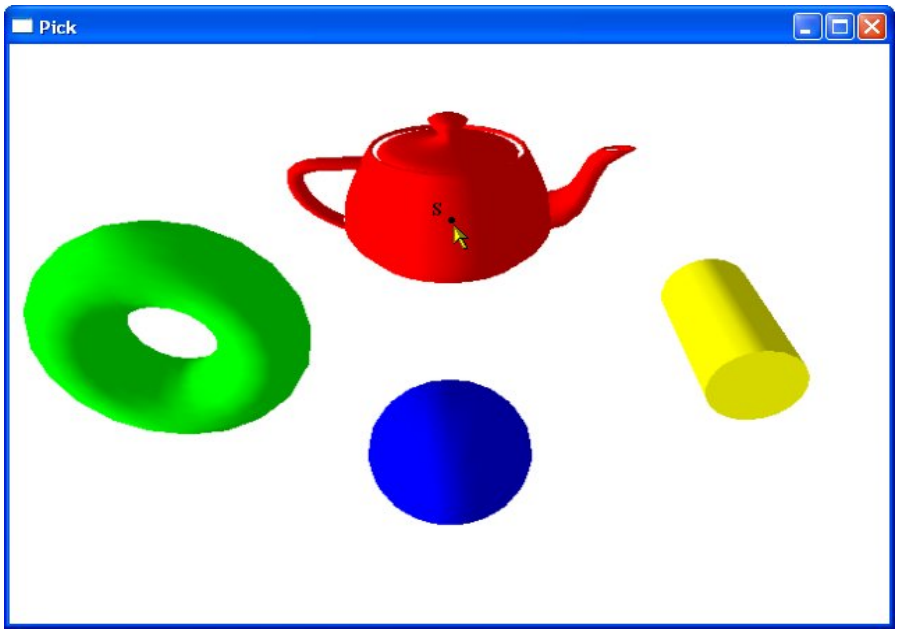

Рис. 15.1. *Пользователь выбирает чайник*

Мы знаем о чайнике то, что его изображение было спроецировано на область экрана внутри которой находится точка **s**. Более правильно будет сказать, что чайник споецирован в окне проекции таким образом, что точка окна проекции **p**, соответствующая точке экрана **s** находится внутри занимаемой им области. Так как эта задача зависит от связи между объектом и его проекцией, решение нам поможет найти рис. 15.2.

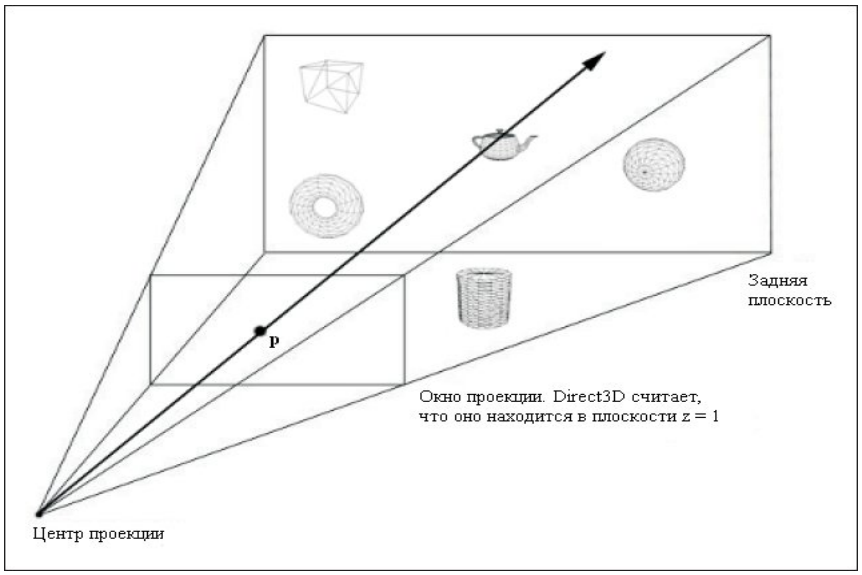

Рис. 15.2. *Луч, проходящий через точку* ***p****, пересекает объект, если точка* ***p*** *находится внутри его проекции. Обратите внимание, что точка* ***p*** *в окне проекции соответствует точке* ***s*** *на экране, по которой щелкнул пользователь*

На рис. 15.2 видно, что луч, начинающийся в начале координат и проходящий через точку **p** пересекает тот объект, внутри которого находится точка **p**, в нашем случае это чайник. Следовательно, вычислив луч выбора мы можем перебрать все объекты сцены и проверить, пересекает ли их луч. Тот объект, который пересекает луч, и будет тем объектом, который выбрал пользователь. Ну и снова скажем, что в нашем примере это чайник.

Итак, в примере у нас есть точка **s** и чайник. В общем случае у нас есть точка экрана по которой щелкнул пользователь. Теперь нам надо вычислить луч выбора и проверить все объекты сцены, не пересекает ли их этот луч. Тот объект, который пересекает луч и будет тем объектом, который выбрал пользователь. Может случиться так, что луч не пересекает никаких объектов. Например, на рис. 15.1 пользователь может не выбирать один из четырех объектов на экране, а щелкнуть по белому фону, и луч выбора не пересечет ни один из объектов. Таким образом мы можем заключить, что если луч не пересекает ни один из объектов сцены, значит пользователь выбрал фон или что-нибудь не представляющее для нас интереса.

Выбор объектов используется во всех играх и трехмерных приложениях. Например, почти всегда пользователь взаимодействует с объектами игрового мира щелкая по их изображениям мышью. Игрок может щелкнуть по врагу, чтобы выстрелить в него, или по предмету, чтобы взять его. Чтобы игра правильно реагировала на действия пользователя, нам надо определить по какому объекту произведен щелчок (это враг или предмет?) и местоположение этого объекта в пространстве (куда должен быть выпущен снаряд или может ли игрок дотянуться до этого предмета?). На все эти вопросы и дает ответ выбор объектов.

Цели

- Изучить реализацию алгоритма выбора объектов и понять как он работает. Выбор объектов можно разделить на четыре этапа:
  - Получение данных точки экрана **s** по которой был сделан щелчок и вычисление соответствующей ей точки **p** в окне проекции.
  - Вычисление луча выбора, который начинается в начале координат и проходит через точку **p**.
  - Преобразование луча выбора и объектов сцены в одну общую систему координат.
  - Определение объекта, пересекаемого лучом выбора. Тот объект, который пересекает луч, и есть тот объект, который выбрал пользователь на экране.

15.1 Преобразование из экранного пространства в окно проекции

Самой первой задачей является преобразование точки на экране в координатную систему окна проекции. Матрица преобразования порта просмотра выглядит так:

$$\begin{bmatrix} \frac{Width}{2} & 0 & 0 & 0 \\ 0 & -\frac{Height}{2} & 0 & 0 \\ 0 & 0 & MaxZ - MinZ & 0 \\ X + \frac{Width}{2} & Y + \frac{Height}{2} & MinZ & 1 \end{bmatrix}$$

Применение данного преобразования к точке окна проекции $\mathbf{p} = (p_x, p_y, p_z)$ дает точку экрана $\mathbf{s} = (s_x, s_y)$:

$$s_x = p_x\left(\frac{Width}{2}\right) + X + \frac{Width}{2}$$

$$s_y = -p_y\left(\frac{Height}{2}\right) + Y + \frac{Height}{2}$$

Обратите внимание, что после преобразования порта просмотра координата Z точки не сохраняется как часть двухмерного изображения, а переносится в буфер глубины.

В нашей ситуации нам известна точка экрана **s**, а мы должны найти точку **p**. Решая приведенные уравнения мы получаем:

$$p_x = \frac{2s_x - 2X - Width}{Width}$$

$$p_y = \frac{-2s_y + 2Y + Height}{Height}$$

Предполагая, что значения X и Y для порта просмотра равны 0, как обычно и бывает, мы можем упростить формулы и получим:

$$p_x = \frac{2s_x}{Width} - 1$$

$$p_y = \frac{-2s_y}{Height} - 1$$

$$p_z = 1$$

По определению окно проекции совпадает с плоскостью $z = 1$; поэтому $p_z = 1$.

Но это еще не все. Матрица проекции масштабирует точки окна проекции, чтобы имитировать различные углы поля зрения. Чтобы получить параметры точки до масштабирования, мы должны инвертировать операцию масштабирования. Для матрицы проекции **P** коэффициенты масштабирования точки по осям X и Y это элементы $\mathbf{P}_{00}$ и $\mathbf{P}_{11}$, так что мы получаем формулы:

$$p_x = \left(\frac{2x}{viewportWidth} - 1\right)\left(\frac{1}{\mathbf{P}_{00}}\right)$$

$$p_y = \left(\frac{-2y}{viewportHeight} + 1\right)\left(\frac{1}{\mathbf{P}_{11}}\right)$$

$$p_z = 1$$

15.2 Вычисление луча выбора

Вспомните, что луч описывается параметрическим уравнением $\mathbf{p}(t) = \mathbf{p}_0 + t\mathbf{u}$, где $\mathbf{p}_0$ — это точка, являющаяся началом луча, а **u** — это вектор, задающий направление луча.

На рис. 15.2 видно, что начало луча является также началом координат пространства вида, так что $\mathbf{p}_0 = (0, 0, 0)$. Если **p** — это точка окна проекции, через которую проходит луч, то вектор направления **u** получаем по формуле $\mathbf{u} = \mathbf{p} - \mathbf{p}_0 = (p_x, p_y, 1) - (0, 0, 0) = \mathbf{p}$.

Приведенный ниже метод вычисляет луч выбора в пространстве вида по заданным координатам x и y точки экранного пространства, по которой был выполнен щелчок:

```
d3d::Ray CalcPickingRay(int x, int y)
{
     float px = 0.0f;
     float py = 0.0f;

     D3DVIEWPORT9 vp;
     Device->GetViewport(&vp);

     D3DXMATRIX proj;
     Device->GetTransform(D3DTS_PROJECTION, &proj);

     px = ((( 2.0f*x) / vp.Width)  - 1.0f) / proj(0, 0);
     py = (((-2.0f*y) / vp.Height) + 1.0f) / proj(1, 1);

     d3d::Ray ray;
     ray._origin = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
     ray._direction = D3DXVECTOR3(px, py, 1.0f);

     return ray;
}
```

где определение **`Ray`** выглядит так:

```
struct Ray
{
     D3DXVECTOR3 _origin;
     D3DXVECTOR3 _direction;
};
```

Мы обновляем файл d3dUtility.h и пространство имен **d3d**, добавляя туда определение **Ray**.

15.3 Преобразование лучей

Луч выбора, который мы вычислили в предыдущем разделе, описан в пространстве вида. Чтобы выполнить проверку пересечения луча и объекта необходимо, чтобы и луч и объект находились в одной и той же системе координат. Вместо того, чтобы преобразовывать все объекты в пространстве вида, зачастую проще преобразовать луч выбора в мировое пространство или даже в локальное пространство объекта.

Мы можем преобразовать луч $\mathbf{r}(t) = \mathbf{p}_0 + t\mathbf{u}$ преобразовав его начальную точку $\mathbf{p}_0$ и вектор направления $\mathbf{u}$ путем умножения на матрицу преобразования. Обратите внимание, что начальная точка преобразуется как точка, а направление обрабатывается как вектор. В рассматриваемом в данной главе примере программы для преобразования лучей реализована следующая функция:

```
void TransformRay(d3d::Ray* ray, D3DXMATRIX* T)
{
     // Преобразование начальной точки луча, w = 1
     D3DXVec3TransformCoord(
          &ray->_origin,
          &ray->_origin,
          T);

     // Преобразование вектора направления луча, w = 0
     D3DXVec3TransformNormal(
          &ray->_direction,
          &ray->_direction,
          T);

     // Нормализация вектора направления
     D3DXVec3Normalize(&ray->_direction, &ray->_direction);
}
```

Функции **D3DXVec3TransformCoord** и **D3DXVec3TransformNormal** получают в качестве параметра трехмерный вектор, но обратите внимание, что функция **D3DXVec3TransformCoord** подразумевает что четвертая компонента вектора $w = 1$. В противоположность ей функция **D3DXVec3TransformNormal** подразумевает, что четвертая компонента вектора $w = 0$. Следовательно, мы используем **D3DXVec3TransformCoord** для преобразования точек, а **D3DXVec3TransformNormal** — для преобразования векторов.

15.4 Пересечение луча и объекта

Поместив луч выбора и объекты в одну систему координат, мы готовы проверить какие объекты пересекает луч. Поскольку объекты представлены сетками с треугольными ячейками, один из возможных способов заключается в следующем. Для каждого объекта мы перебираем элементы его списка граней и проверяем не пересекает ли луч какой-нибудь из этих треугольников. Если да, значит луч попал в тот объект, которому принадлежит треугольник.

Однако, выполнение проверки пересечения с лучом для каждого треугольника сцены требует много времени на вычисления. Более быстрый, хотя и менее точный метод — представить каждый объект с помощью ограничивающей сферы. Тогда мы выполняем проверку пересечения луча и ограничивающей сферы и тот объект, чью ограничивающую сферу пересекает луч, считается выбранным.

ПРИМЕЧАНИЕ Луч выбора может пересекать несколько объектов. В этом случае выбранным считается самый близкий к камере объект, потому что дальние объекты закрываются им.

Зная центральную точку **c** и радиус **r** сферы, мы можем проверить находится ли точка **p** на поверхности сферы с помощью следующей простой формулы:

$$\left| \mathbf{p} - \mathbf{c} \right| - r = 0$$

где **p** — это точка на сфере, если условие выполняется (рис. 15.3).

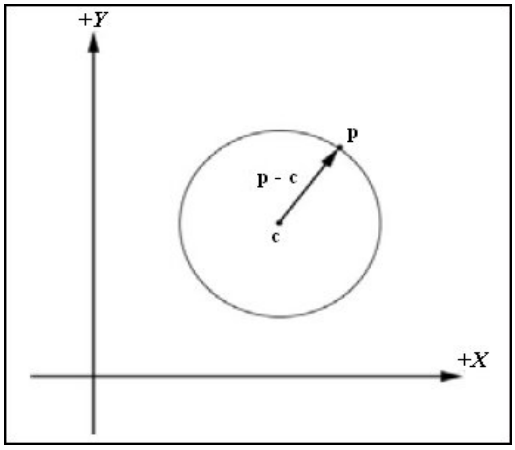

Рис. 15.3. *Длина вектора* ***p*** *–* ***c****, обозначаемая как |****p*** *–* ***c****|, равна радиусу сферы, если точка* ***p*** *лежит на поверхности сферы. Обратите внимание, что на иллюстрации для простоты изображен круг, но идея работает и в трех измерениях*

Чтобы определить, пересекает ли луч $\mathbf{p}(t) = \mathbf{p}_0 + t\mathbf{u}$ сферу и, если да, то где, мы подставляем формулу луча в уравнение сферы и ищем значение параметра t, удовлетворяющее уравнению сферы, что позволит нам найти точки пересечения.

Подставляем формулу луча в уравнение сферы:

$$\left| \mathbf{p}(t) - \mathbf{c} \right| - r = 0$$

$$\left| \mathbf{p}_0 + t\mathbf{u} - \mathbf{c} \right| - r = 0$$

и получаем квадратное уравнение:

$$At^2 + Bt + C = 0$$

где $A = \mathbf{u} \cdot \mathbf{u}$, $B = 2(\mathbf{u} \cdot (\mathbf{p}_0 - \mathbf{c}))$ и $C = (\mathbf{p}_0 - \mathbf{c}) \cdot (\mathbf{p}_0 - \mathbf{c}) - r^2$. Если вектор **u** нормализован, то $A = 1$.

Предполагая, что вектор **u** нормализован, получаем решения для t_0 и t_1:

$$t_0 = \frac{-B + \sqrt{B^2 - 4C}}{2} \quad t_1 = \frac{-B - \sqrt{B^2 - 4C}}{2}$$

На рис. 15.4 показаны возможные результаты для t_0 и t_1 и объяснено, что эти результаты означают с точки зрения геометрии.

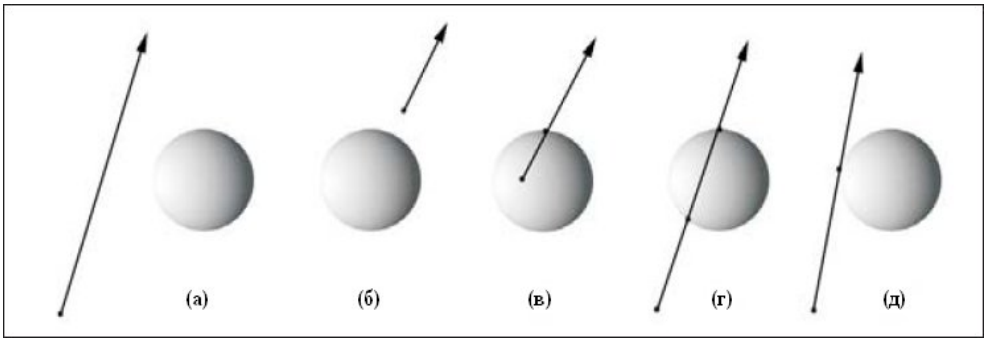

Рис. 15.4. *а) Луч проходит мимо сферы; и t_0 и t_1 мнимые числа б) Луч находится за сферой; и t_0 и t_1 отрицательные числа. в) Луч начинается внутри сферы; одно из решений положительное, а другое — отрицательное. Положительное решение соответствует единственной точке пересечения. г) Луч пересекает сферу; и t_0 и t_1 положительные числа. д) Луч касается сферы в единственной точке; в этом случае оба решения положительны и $t_0 = t_1$.*

Приведенный ниже метод возвращает **`true`**, если переданный в первом параметре луч пересекает переданную во втором параметре сферу. Если луч проходит мимо сферы, метод возвращает **`false`**:

```
bool PickApp::raySphereIntersectionTest(Ray* ray,
                                        BoundingSphere* sphere)
{
     D3DXVECTOR3 v = ray->_origin - sphere->_center;

     float b = 2.0f * D3DXVec3Dot(&ray->_direction, &v);
     float c = D3DXVec3Dot(&v, &v) -
               (sphere->_radius * sphere->_radius);


     // Находим дискриминант
     float discriminant = (b * b) - (4.0f * c);

     // Проверяем на мнимые числа
     if(discriminant < 0.0f)
```

```
            return false;

        discriminant = sqrtf(discriminant);

        float s0 = (-b + discriminant) / 2.0f;
        float s1 = (-b - discriminant) / 2.0f;

        // Если есть решение >= 0, луч пересекает сферу
        if(s0 >= 0.0f || s1 >= 0.0f)
            return true;

        return false;
}
```

Конечно, мы уже показывали объявление структуры **`BoundingSphere`**, но для удобства приведем его здесь еще раз:

```
struct BoundingSphere
{
    BoundingSphere();

    D3DXVECTOR3 _center;
    float _radius;
};
```

15.5 Пример приложения: выбор объекта

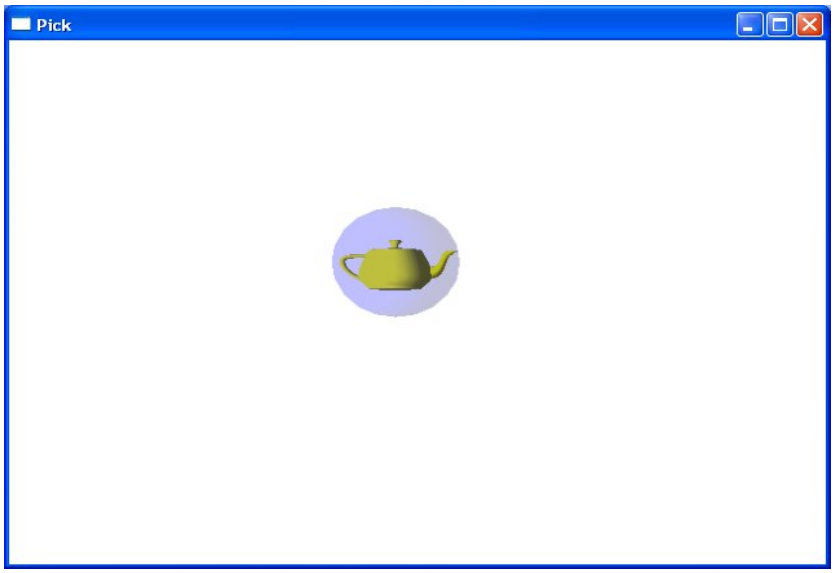

Рис. 15.5. *Окно примера к данной главе*

На рис. 15.5 показано окно приложения, созданного для данной главы. Чайник перемещается по экрану, а вы должны попытаться щелкнуть по нему мышкой. Если вы попали в ограничивающую сферу чайника, на экран будет выведено

сообщающее об этом диалоговое окно. Мы обрабатываем событие щелчка кнопки мыши проверяя сообщение **WM_LBUTTONDOWN**:

```
case WM_LBUTTONDOWN:
     // Вычисляем луч в пространстве вида на основании
     // координат указателя мыши в момент щелчка
     d3d::Ray ray = CalcPickingRay(LOWORD(lParam), HIWORD(lParam));

     // Преобразуем луч в мировое пространство
     D3DXMATRIX view;
     Device->GetTransform(D3DTS_VIEW, &view);

     D3DXMATRIX viewInverse;
     D3DXMatrixInverse(&viewInverse, 0, &view);

     TransformRay(&ray, &viewInverse);

     // Проверяем попадание
     if(RaySphereIntTest(&ray, &BSphere))
          ::MessageBox(0, "Hit!", "HIT", 0);

     break;
```

15.6 Итоги

- Выбор объекта — это техника используемая для определения трехмерного объекта, соответствующего отображаемой на экране двухмерной проекции по которой щелкнул пользователь.
- Луч выбора формируется путем создания луча, начинающегося в начале координат пространства вида и проходящего через точку окна проекции, соответствующую той точке экрана, по которой щелкнул пользователь.
- Мы можем преобразовать луч $\mathbf{r}(t) = \mathbf{p}_0 + t\mathbf{u}$ преобразовав по отдельности его начальную точку $\mathbf{p}_0$ и вектор направления $\mathbf{u}$ с помощью матрицы преобразования. Обратите внимание, что начало луча преобразуется как точка ($w = 1$), а направление — как вектор ($w = 0$).
- Чтобы проверить, пересекает ли луч объект, мы можем проверить пересекает ли луч какую-нибудь из треугольных граней объекта или проверить пересекает ли луч ограничивающий объем объекта, например, ограничивающую сферу.

Часть IV
Шейдеры и эффекты

До сих пор мы достигали желаемого эффекта путем изменения конфигурации состояний устройства таких как преобразования, освещение, текстуры и режимы визуализации. Хотя набор поддерживаемых конфигураций обеспечивает достаточную гибкость, мы ограничены предопределенными операциями (отсюда и название «фиксированный конвейер функций»).

Главная тема этой части — вершинные и пиксельные шейдеры, которые заменяют части фиксированного конвейера функций реализуемыми нами программами, которые называются *шейдеры* (*shader*). Шейдеры полностью программируемы и позволяют нам реализовать техники, которые отсутствуют в фиксированном конвейере функций. В результате количество доступных техник значительно расширяется. Программируемые части конвейера визуализации обычно называют программируемым конвейером. Ниже приведен краткий обзор глав этой части книги.

Глава 16, «Введение в высокоуровневый язык шейдеров» — В этой главе мы познакомим вас с высокоуровневым языком шейдеров (High-Level Shading Language, HLSL). Этот язык мы будем использовать в данной книге для написания программ вершинных и пиксельных шейдеров.

Глава 17, «Знакомство с вершинными шейдерами» — В этой главе мы узнаем что такое вершинные шейдеры и как их создать и использовать в Direct3D. Работа с вершинными шейдерами иллюстрируется исследованием реализации техники мультипликационного затенения.

Глава 18, «Знакомство с пиксельными шейдерами» — В этой главе мы узнаем что такое пиксельные шейдеры и как их создать и использовать в Direct3D. В качестве примера в главе будет рассмотрена реализация мультитекстурирования с использованием пиксельных шейдеров.

Глава 19, «Каркас эффектов» — В этой главе мы обсудим каркас эффектов Direct3D. В главе объясняется назначение каркаса эффектов, структуру и синтаксис файлов эффектов и создание и использование файлов эффектов в приложениях Direct3D.

Глава 16
Введение в высокоуровневый язык шейдеров

В этой главе мы рассмотрим высокоуровневый язык программирования шейдеров (High-Level Shading Language, HLSL), который будем использовать для программирования вершинных и пиксельных шейдеров в следующих трех главах. Если коротко, вершинные и пиксельные шейдеры — это небольшие программы, которые вы пишете сами, и которые выполняются процессором видеокарты, заменяя часть функций фиксированного конвейера. Возможность заменять функции фиксированного конвейера собственными программами шейдеров открывает перед нами необозримые возможности реализации различных визуальных эффектов. Мы больше не ограничены предопределенными «фиксированными» операциями.

Для того, чтобы написать программу шейдера, нам необходим позволяющий сделать это язык программирования. В DirectX 8.*x* программы шейдеров можно было писать только на низкоуровневом языке ассемблера шейдеров. К счастью, нам больше не требуется писать шейдеры на языке ассемблера, поскольку DirectX 9 предоставляет предназначенный для написания шейдеров высокоуровневый язык программирования шейдеров (High-Level Shading Language). Использование для написания программ шейдеров HLSL вместо языка ассемблера дает те же преимущества, что и использование высокоуровневого языка, такого как C++, вместо ассемблера при написании приложений:

- Увеличивается производительность труда — писать программы на высокоуровневом языке быстрее и проще, чем на низкоуровневом. Мы можем больше времени уделить разработке алгоритмов, а не кодированию.
- Улучшается читаемость программ — программы на высокоуровневом языке проще читать, а значит их проще отлаживать и поддерживать.
- Компиляторы часто генерируют боле эффективный ассемблерный код, чем тот, который вы сможете написать сами.
- Используя компилятор HLSL мы можем компилировать код для любой доступной версии шейдеров. Если мы пользуемся языком ассемблера, то должны портировать код для каждой версии, поддержка которой нам требуется.

Синтаксис HLSL очень похож на синтакс C и C++, так что изучить этот язык будет достаточно просто.

Учтите, что если ваша видеокарта не поддерживает вершинные и пиксельные шейдеры, то для использующих шейдеры программ в коде надо выбирать устройство REF. Использование устройства REF означает, что примеры работы с шейдерами будут выполняться очень медленно, но при этом на экране будет отображаться корректный результат, позволяющий убедиться в правильности написанного кода.

ПРИМЕЧАНИЕ Вершинные шейдеры могут эмулироваться программно, если при создании устройства включить программную обработку вершин, указав флаг **`D3DCREATE_SOFTWARE_VERTEXPROCESSING`**.

Цели

- Узнать как написать и скомпилировать программу шейдера на HLSL.
- Изучить, как выполняется обмен данными между приложением и программой шейдера.
- Познакомиться с синтаксисом, типами и встроенными функциями HLSL.

16.1 Пишем шейдер на HLSL

Мы можем написать код нашего HLSL-шейдера непосредственно в исходном коде приложения в виде длинной строки символов. Однако более удобный и правильный подход — разделить код шейдеров и код приложения. Поэтому мы будем писать наши шейдеры в программе Notepad и сохранять их как обычные текстовые файлы ASCII. Затем для компиляции наших шейдеров мы воспользуемся функцией **`D3DXCompileShaderFromFile`** (раздел 16.2.2).

В качестве примера, рассмотрим приведенный ниже простой вершинный шейдер, написанный на HLSL и сохраненный в текстовый файл с именем Transform.txt. Полный код проекта находится в папке с именем Transform, расположенной в сопроводительных файлах к данной главе. Вершинный шейдер преобразует вершину путем применения комбинации матриц вида и проекции а также присваивает рассеиваемой составляющей цвета вершины синий цвет.

ПРИМЕЧАНИЕ В качестве примера мы здесь рассматриваем вершинный шейдер, но вам пока не надо беспокоиться о том, что должен делать вершинный шейдер, так как этой теме посвящена следующая глава. Сейчас нашей целью является знакомство с синтаксисом и форматом программ на HLSL.

```
//////////////////////////////////////////////////////////////////////
//
// Файл: transform.txt
//
// Автор: Фрэнк Д. Луна (C) All Rights Reserved
//
// Система: AMD Athlon 1800+ XP, 512 DDR, Geforce 3, Windows XP,
// MSVC++ 7.0
//
// Описание: Вершинный шейдер, преобразующий вершину с помощью
// комбинации матриц преобразования вида и проекции и устанавливающий
// для вершины синий цвет
//
//////////////////////////////////////////////////////////////////////

//
// Глобальные переменные
//

// Глобальная переменная для хранения комбинации
// матриц преобразования вида и проекции.
// Мы инициализируем эту переменную в приложении.
matrix ViewProjMatrix;

// Инициализация глобального вектора для синего цвета
vector Blue = {0.0f, 0.0f, 1.0f, 1.0f};
```

```
//
// Структуры
//
// Входная структура описывает вершины, которые будут
// передаваться в шейдер. Здесь входная вершина содержит
// только данные о местоположении.
struct VS_INPUT
{
     vector position : POSITION;
};
// Выходная структура описывает вершину, которая
// возвращается шейдером. Здесь выходная вершина
// содержит данные о местоположении и цвет
struct VS_OUTPUT
{
     vector position : POSITION;
     vector diffuse : COLOR;
};
//
// Главная Точка Входа. Обратите внимание,
// что функция получает в своем параметре копию
// входной вершины и возвращает копию вычисленной
// выходной вершины.
//
VS_OUTPUT Main(VS_INPUT input)
{
     // Обнуляем данные выходной вершины
     VS_OUTPUT output = (VS_OUTPUT)0;
     // Преобразование пространства вида и проекция
     output.position = mul(input.position, ViewProjMatrix);
     // Делаем рассеиваемую составляющую цвета синей
     output.diffuse = Blue;
     // Возвращаем спроецированную и окрашенную вершину
     return output;
}
```

16.1.1 Глобальные переменные

Сначала мы объявляем две глобальные переменные:

```
matrix ViewProjMatrix;
vector Blue = {0.0f, 0.0f, 1.0f, 1.0f};
```

Первая переменная, **`ViewProjMatrix`**, относится к типу **`matrix`**, который представляет матрицы 4 × 4 и является встроенным типом HLSL. Эта переменная хранит комбинацию матриц вида и проекции и, следовательно, описывает оба эти преобразования. Благодаря комбинированию преобразований мы обойдемся одной операцией умножения вектора на матрицу вместо двух. Обратите внимание, что нигде в исходном коде шейдера нет инициализации этой переменной. Это объясняется тем, что мы инициализируем данную переменную

из приложения, а не в шейдере. Обмен данными между приложением и программой шейдера является одной из наиболее часто используемых операций, и ее исследованию посвящен раздел 16.2.1.

Вторая переменная, **Blue**, относится к встроенному типу **vector**, который представляет четырехмерный вектор. Мы просто инициализируем его компоненты для синего цвета, рассматривая его как цветовой вектор RGBA.

16.1.2 Входная и выходная структуры

За объявлением глобальных переменных следует объявление двух специальных структур, которые мы будем называть *входной* (*input*) и *выходной* (*output*) структурами. Для вершинных шейдеров эти структуры описывают данные вершины, которые соответственно, получает и возвращает наш шейдер.

```
struct VS_INPUT
{
     vector position : POSITION;
};

struct VS_OUTPUT
{
     vector position : POSITION;
     vector diffuse : COLOR;
};
```

ПРИМЕЧАНИЕ Входная и выходная структура пиксельного шейдера описывают данные пикселя.

В рассматриваемом примере вершина, поступающая на вход шейдера содержит только данные о местоположении. Возвращаемая нашим шейдером вершина содержит сведения о местоположении а также данные цвета.

Синтаксическая конструкция с двоеточием применяется для указания способа использования переменной. Это похоже на поле настраиваемого формата вершин (FVF) в структуре данных вершины. Например, во входной структуре данных **VS_INPUT**, у нас есть член

```
vector position : POSITION;
```

Конструкция **: POSITION** означает, что переменная **position** типа **vector** используется для описания местоположения передаваемой шейдеру вершины. В качестве другого примера можно рассмотреть описание члена структуры **VS_OUTPUT**:

```
vector diffuse : COLOR;
```

Здесь **: COLOR** означает, что переменная **diffuse** типа **vector** применяется для описания цвета возвращаемой шейдером вершины. О доступных идентификаторах способов использования переменных, применяемых для вершинных и пиксельных шейдеров, мы поговорим в следующих двух главах.

ПРИМЕЧАНИЕ С низкоуровневой точки зрения данная конструкция связывает переменную шейдера с регистром аппаратуры. Входные переменные связываются с входными регистрами, а выходные — с выходными. Например, член **position** структуры **VS_INPUT** связан с входным регистром местоположения вершины. Аналогично, член **diffuse** связан с выходным регистром цвета вершины.

16.1.3 Точка входа

Подобно программам на C++, у каждой программы на HLSL есть точка входа. В нашем примере вершинного шейдера мы назвали являющуюся точкой входа функцию **Main**, но это не обязательно. В качестве точки входа шейдера может использоваться любая функция, независимо от ее имени. Учтите, что у этой функции должны быть входные параметры, которые используются для передачи данных исходной вершины в шейдер. Кроме того, функция должна возвращать выходную структуру, применяемую для возврата вершины, обработанной нашим шейдером.

```
VS_OUTPUT Main(VS_INPUT input)
{
```

ПРИМЕЧАНИЕ В действительности вы не обязаны использовать входные и выходные структуры. Например, в особенности для пиксельных шейдеров, вы часто будете встречаться с синтаксисом, похожим на приведенный ниже:

```
float4 Main(in float2 base : TEXCOORD0,
            in float2 spot : TEXCOORD1,
            in float2 text : TEXCOORD2) : COLOR
{
     ...
}
```

Параметры передаются в шейдер; в данном примере мы передаем шейдеру три набора координат текстуры. Шейдер возвращает единственное значение цвета, на что указывает констркукия **: COLOR** следующая за сигнатурой функции. Это определение эквивалентно следующему:

```
struct INPUT
{
     float2 base : TEXCOORD0;
     float2 spot : TEXCOORD1;
     float2 text : TEXCOORD2;
};
struct OUTPUT
{
     float4 c : COLOR;
};

OUTPUT Main(INPUT input)
{
     ...
}
```

Код функции, являющейся входной точкой, отвечает за вычисление данных возвращаемой вершины на основе полученных данных исходной вершины. Рассматриваемый в примере шейдер просто преобразует координаты вершины в пространство вида и пространство проекции, устанавливает для вершины синий цвет и возвращает полученную в результате вершину. Сперва мы создаем экземпляр выходной структуры **VS_OUTPUT** и присваиваем всем ее членам 0.

```
VS_OUTPUT output = (VS_OUTPUT)0; // обнуляем все члены
```

Затем наш шейдер выполняет преобразование координат исходной вершины, умножая ее на переменную **ViewProjMatrix** с помощью функции **mul**, которая является встроенной функцией, выполняющей операции умножения вектора на матрицу и умножения матрицы на матрицу. Преобразованный вектор местоположения вершины мы сохраняем в члене **position** экземпляра выходной структуры данных:

```
// Преобразование и проекция
output.position = mul(input.position, ViewProjMatrix);
```

Затем мы устанавливаем член данных, задающий рассеиваемую составляющую цвета, равной вектору **Blue**:

```
// Делаем рассеиваемую составляющую цвета синей
output.diffuse = Blue;
```

И, наконец, мы возвращаем полученную вершину:

```
return output;
}
```

16.2 Компиляция шейдеров на HLSL

16.2.1 Таблица констант

В каждом шейдере есть таблица констант, используемая для хранения его переменных. Библиотека D3DX обеспечивает приложению доступ к таблице констант шейдера через интерфейс **ID3DXConstantTable**. Через этот интерфейс мы можем устанавливать значения переменных шейдера из кода нашего приложения.

Сейчас мы приведем сокращенный список методов, реализуемых интерфейсом **ID3DXConstantTable**. Если вам необходим полный список, обратитесь к документации Direct3D.

16.2.1.1. Получение дескриптора константы

Чтобы установить значение какой-нибудь переменной шейдера из кода нашего приложения, необходим способ сослаться на эту переменную. Для этой цели

применяется тип **D3DXHANDLE**. Приведенный ниже метод возвращает значение типа **D3DXHANDLE**, указывающее на переменную шейдера с заданным именем:

```
D3DXHANDLE ID3DXConstantTable::GetConstantByName(
     D3DXHANDLE hConstant, // область видимости
     LPCSTR pName          // имя
);
```

- **hConstant** — Значение **D3DXHANDLE** задающее родительскую структуру запрашиваемой переменной, определяющую время ее жизни. Например, если мы хотим получить дескриптор отдельного члена объявленной в шейдере структуры данных, то здесь нам надо указать дескриптор этой структуры. Если мы получаем дескриптор переменной самого верхнего уровня, в этом параметре передается 0.
- **pName** — указанное в исходном коде шейдера имя переменной, для которой мы получаем дескриптор.

Например, если имя переменной в коде шейдера **ViewProjMatrix** и это переменная верхнего уровня, то для получения дескриптора следует написать:

```
// Получение дескриптора переменной шейдера ViewProjMatrix
D3DXHANDLE h0;
h0 = ConstTable->GetConstantByName(0, "ViewProjMatrix");
```

16.2.1.2. Установка констант

Как только наше приложение получило значение **D3DXHANDLE**, ссылающееся на требуемую переменную в коде шейдера, мы можем установить значение этой переменной из нашего приложения с помощью метода **ID3DXConstantTable::Set*XXX***, где ***XXX*** заменяется на название типа переменной, значение которой устанавливается. Например, если мы хотим установить значения массива векторов, следует воспользоваться методом **SetVectorArray**.

Общий синтаксис всех методов **ID3DXConstantTable::Set*XXX*** выглядит так:

```
HRESULT ID3DXConstantTable::SetXXX(
     LPDIRECT3DDEVICE9 pDevice,
     D3DXHANDLE hConstant,
     XXX value
);
```

- **pDevice** — Указатель на устройство с которым связана таблица констант.
- **hConstant** — Дескриптор, ссылающийся на переменную, значение которой мы устанавливаем.
- **value** — Присваиваемое переменной значение, где ***XXX*** заменяется на название типа переменной, значение которой мы устанавливаем. Для

некоторых значений (**bool**, **int**, **float**) мы передаем само значение, а для других (векторы, матрицы, структуры) — ссылку на значение.

Если мы инициализируем массив, то у метода **Set*XXX*** появляется дополнительный четвертый параметр, задающий количество элементов массива. Например, прототип метода для установки значений массива четырехмерных векторов, выглядит так:

```
HRESULT ID3DXConstantTable::SetVectorArray(
     LPDIRECT3DDEVICE9 pDevice,  // связанное устройство
     D3DXHANDLE hConstant,       // дескриптор переменной шейдера
     CONST D3DXVECTOR4* pVector, // указатель на массив
     UINT Count                  // количество элементов массива
);
```

Приведенный ниже список описывает типы, которые мы можем инициализировать с помощью интерфейса **ID3DXConstantTable**. Подразумевается, что корректное устройство (**Device**) и корректный дескриптор переменной (**handle**) уже получены.

- **SetBool** — используется для установки логических значений. Пример вызова:

```
bool b = true;
ConstTable->SetBool(Device, handle, b);
```

- **SetBoolArray** — Используется для установки массива логических значений. Пример вызова:

```
bool b[3] = {true, false, true};
ConstTable->SetBoolArray(Device, handle, b, 3);
```

- **SetFloat** — Используется для установки значения с плавающей точкой. Пример вызова:

```
float f = 3.14f;
ConstTable->SetFloat(Device, handle, f);
```

- **SetFloatArray** — Используется для установки массива значений с плавающей точкой. Пример вызова:

```
float f[2] = {1.0f, 2.0f};
ConstTable->SetFloatArray(Device, handle, f, 2);
```

- **SetInt** — Используется для установки целочисленного значения. Пример вызова:

```
int x = 4;
ConstTable->SetInt(Device, handle, x);
```

- **SetIntArray** — Используется для установки массива целых чисел. Пример вызова:

```
int x[4] = {1, 2, 3, 4};
ConstTable->SetIntArray(Device, handle, x, 4);
```

- **SetMatrix** — Используется для установки матрицы 4 × 4. Пример вызова:

```
D3DXMATRIX M(...);
ConstTable->SetMatrix(Device, handle, &M);
```

- **SetMatrixArray** — Используется для установки массива матриц 4 × 4. Пример вызова:

```
D3DXMATRIX M[4];

// ...Инициализация матриц

ConstTable->SetMatrixArray(Device, handle, M, 4);
```

- **SetMatrixPointerArray** — Используется для установки массива указателей на матрицы 4 × 4. Пример вызова:

```
D3DXMATRIX* M[4];

// ...Выделение памяти и инициализация указателей

ConstTable->SetMatrixPointerArray(Device, handle, M, 4);
```

- **SetMatrixTranspose** — используется для установки транспонированной матрицы 4 × 4. Пример вызова:

```
D3DXMATRIX M(...);
D3DXMatrixTranspose(&M, &M);
ConstTable->SetMatrixTranspose(Device, handle, &M);
```

- **SetMatrixTransposeArray** — Используется для установки массива транспонированных матриц 4 × 4. Пример вызова:

```
D3DXMATRIX M[4];

// ...Инициализация матриц и их транспонирование

ConstTable->SetMatrixTransposeArray(Device, handle, M, 4);
```

- **SetMatrixTransposePointerArray** — Используется для установки массива указателей на транспонированные матрицы 4 × 4. Пример вызова:

```
D3DXMATRIX* M[4];

// ...Выделение памяти, инициализация указателей
//    и транспонирование

ConstTable->SetMatrixTransposePointerArray(Device, handle,
                                                    M, 4);
```

- **SetVector** — Используется для установки переменной типа **D3DXVECTOR4**. Пример вызова:

```
D3DXVECTOR4 v(1.0f, 2.0f, 3.0f, 4.0f);
ConstTable->SetVector(Device, handle, &v);
```

- **SetVectorArray** — Используется для установки массива векторов. Пример вызова:

```
D3DXVECTOR4 v[3];

// ...Инициализация векторов

ConstTable->SetVectorArray(Device, handle, v, 3);
```

- **SetValue** — Используется для установки значения произвольного размера, например, структуры. В приведенном примере мы используем **SetValue** для установки значений **D3DXMATRIX**:

```
D3DXMATRIX M(...);
ConstTable->SetValue(Device, handle, (void*)&M, sizeof(M));
```

16.2.1.3. Установка значений по умолчанию для констант

Приведенный ниже метод присваивает всем константам значения по умолчанию, то есть те значения, которые были заданы при объявлении переменных. Метод должен вызываться один раз при инициализации данных приложения.

```
HRESULT ID3DXConstantTable::SetDefaults(
     LPDIRECT3DDEVICE9 pDevice
);
```

- **pDevice** — Указатель на связанное с таблицей констант устройство.

16.2.2 Компиляция HLSL-шейдера

Мы можем скомпилировать шейдер, код которого хранится в текстовом файле, с помощью следующей функции:

```
HRESULT D3DXCompileShaderFromFile(
     LPCSTR               pSrcFile,
     CONST D3DXMACRO*     pDefines,
     LPD3DXINCLUDE        pInclude,
     LPCSTR               pFunctionName,
     LPCSTR               pTarget,
     DWORD                Flags,
     LPD3DXBUFFER*        ppShader,
     LPD3DXBUFFER*        ppErrorMsgs,
     LPD3DXCONSTANTTABLE* ppConstantTable
);
```

- **pSrcFile** — Имя текстового файла, содержащего исходный код шейдера, который вы хотите скомпилировать.

- **pDefines** — Необязательный параметр, и в данной книге мы всегда будем указывать в нем **null**.
- **pInclude** — Указатель на интерфейс **ID3DXInclude**. Этот интерфейс разработан для тех приложений, которым требуется переопределить устанавливаемое по умолчанию поведение включения. В общем случае поведение по умолчанию замечательно работает и поэтому мы игнорируем данный параметр, передавая в нем **null**.
- **pFunctionName** — Строка, задающая имя функции, являющейся точкой входа. Например, если точкой входа шейдера является функция с именем **Main**, мы должны передать в этом параметре строку «Main».
- **pTarget** — Строка, задающая версию шейдеров для которой будет компилироваться исходный код HLSL. Для вершинных шейдеров доступны версии vs_1_1, vs_2_0, vs_2_sw. Для пиксельных шейдеров доступны версии ps_1_1, ps_1_2, ps_1_3, ps_1_4, ps_2_0, ps_2_sw. Например, если мы хотим чтобы наш вершинный шейдер был скомпилирован для версии 2.0, надо указать в этом параметре vs_2_0.

ПРИМЕЧАНИЕ	Возможность компиляции для различных версий шейдеров — это одно из основных преимуществ использования HLSL вместо языка ассемблера. Работая с HLSL мы можем моментально портировать шейдер для другой версии просто перекомпилировав его. Если же используется ассемблер, то придется вручную изменять код.

- **Flags** — Необязательные флаги компиляции; если флаги не нужны, укажите 0. Можно использовать следующие значения:
  - **D3DXSHADER_DEBUG** — Приказывает компилятору включать в скомпилированный файл отладочную информацию.
  - **D3DXSHADER_SKIPVALIDATION** — Приказывает компилятору не выполнять проверку корректности кода. Этот флаг следует использовать только при работе с теми шейдерами в правильности кода которых вы абсолютно уверены.
  - **D3DXSHADER_SKIPOPTIMIZATION** — Приказывает компилятору не выполнять оптимизацию кода. Обычно этот флаг используется при отладке, когда вы не хотите, чтобы компилятор вносил какие-либо изменения в код.
- **ppShader** — Возвращает указатель на интерфейс **ID3DXBuffer**, который содержит скомпилированный код шейдера. Этот скомпилированный код затем передается в параметре другой функции, которая выполняет фактическое создание вершинного или пиксельного шейдера.

- **ppErrorMsgs** — Возвращает указатель на интерфейс **ID3DXBuffer**, содержащий строку с кодами обнаруженых при компиляции ошибок и их описанием.
- **ppConstantTable** — Возвращает указатель на интерфейс **ID3DXConstantTable**, содержащий данные таблицы констант шейдера.

Вот пример вызова функции **D3DXCompileShaderFromFile**:

```
//
// Компиляция шейдера
//
ID3DXConstantTable* TransformConstantTable = 0;
ID3DXBuffer* shader = 0;
ID3DXBuffer* errorBuffer = 0;

hr = D3DXCompileShaderFromFile(
      "transform.txt",  // имя файла шейдера
      0,
      0,
      "Main",           // имя точки входа
      "vs_2_0",         // версия шейдеров
      D3DXSHADER_DEBUG, // компиляция для отладки
      &shader,
      &errorBuffer,
      &TransformConstantTable);

// Выводим сообщения об ошибках
if(errorBuffer)
{
     ::MessageBox(0, (char*)errorBuffer->GetBufferPointer(), 0, 0);
     d3d::Release<ID3DXBuffer*>(errorBuffer);
}

if(FAILED(hr))
{
     ::MessageBox(0, "D3DXCreateEffectFromFile() - FAILED", 0, 0);
     return false;
}
```

16.3 Типы переменных

ПРИМЕЧАНИЕ Помимо описываемых в следующих разделах типов в HLSL также есть несколько встроенных объектных типов (например, объект текстуры). Однако, эти объектные типы используются в основном в каркасе эффектов и поэтому мы отложим их обсуждение до главы 19.

16.3.1 Скалярные типы

HLSL поддерживает следующие скалярные типы:

- **bool** — Логическое значение «истина» или «ложь». Обратите внимание, что в HLSL есть ключевые слова **true** и **false**.
- **int** — 32-разрядное целое со знаком.
- **half** — 16-разрядное число с плавающей точкой.
- **float** — 32-разрядное число с плавающей точкой.
- **double** — 64-разрядное число с плавающей точкой.

ПРИМЕЧАНИЕ Некоторые аппаратные платформы не поддерживают типы **int**, **half** и **double**. В таком случае эти типы эмулируются с помощью **float**.

16.3.2 Векторные типы

HLSL поддерживает следующие встроенные векторные типы:

- **vector** — Четырехмерный вектор, каждая компонента которого имеет тип **float**.
- **vector<T, n>** — *n*-мерный вектор, каждая компонента которого относится к скалярному типу **T**. Размерность *n* может быть от 1 до 4. Вот пример двухмерного вектора с компонентами типа **double**:

```
vector<double, 2> vec2;
```

Доступ к отдельным компонентам вектора осуществляется с использованием синтаксиса доступа к элементу массива по индексу. Например, чтобы установить значение *i*-ой компоненты вектора **vec**, следует написать:

```
vec[i] = 2.0f;
```

Кроме того, мы можем обращаться к компонентам вектора **vec** как к членам структуры, используя предопределенные имена компонентов **x**, **y**, **z**, **w**, **r**, **g**, **b** и **a**.

```
vec.x = vec.r = 1.0f;
vec.y = vec.g = 2.0f;
vec.z = vec.b = 3.0f;
vec.w = vec.a = 4.0f;
```

Имена **r**, **g**, **b** и **a** ссылаются на те же самые компоненты, что и имена **x**, **y**, **z** и **w**, соответственно. Когда вектор используется для представления цвета, нотация RGBA более предпочтительна, поскольку подчеркивает тот факт, что вектор содержит цветовые значения, а не координаты.

Помимо этого, мы можем пользоваться следующими предопределенными типами для представления двухмерных, трехмерных и четырехмерных векторов соответственно:

```
float2 vec2;
float3 vec3;
float4 vec4;
```

Возьмем вектор **u** = (u_x, u_y, u_z, u_w) и предположим, что мы хотим скопировать компоненты вектора **u** в вектор **v**, чтобы получить **v** = (u_x, u_y, u_y, u_w). Первое, что приходит на ум, скопировать каждую компоненту **u** в соответствующую компоненту **v**. Однако, HLSL предоставляет специальный синтаксис для таких операций копирования с изменением последовательности, называемый *перенос по адресам* (*swizzles*):

```
vector u = {1.0f, 2.0f, 3.0f, 4.0f};
vector v = {0.0f, 0.0f, 5.0f, 6.0f};
v = u.xyyw; // v = {1.0f, 2.0f, 2.0f, 4.0f}
```

При копировании векторов мы не обязаны копировать все их компоненты. Например, мы можем скопировать только компоненты x и y, как показано в приведенном ниже фрагменте кода:

```
vector u = {1.0f, 2.0f, 3.0f, 4.0f};
vector v = {0.0f, 0.0f, 5.0f, 6.0f};
v.xy = u; // v = {1.0f, 2.0f, 5.0f, 6.0f}
```

16.3.3 Матричные типы

HLSL предоставляет следующие встроенные матричные типы:

- **matrix** — Матрица 4 × 4 все элементы которой имеют тип **float**.
- **matrix<T, m, n>** — Матрица размера *m* × *n*, каждый элемент которой относится к скалярному типу **T**. Размеры матрицы *m* и *n* могут принимать значения от 1 до 4. Вот пример матрицы целых чисел, размером 2 × 2:

  ```
  matrix<int, 2, 2> m2x2;
  ```

Кроме того, мы можем объявить матрицу m × n, где *m* и *n* — числа в диапазоне от 1 до 4, используя следующий синтаксис:

```
floatmxn matmxn;
```

Примеры:

```
float2x2 mat2x2;
float3x3 mat3x3;
float4x4 mat4x4;
float2x4 mat2x4;
```

ПРИМЕЧАНИЕ Тип не обязательно должен быть **float** — можно использовать и другие типы. Например, мы можем объявлять целочисленные матрицы:

```
int2x2 i2x2;
int2x2 i3x3;
int2x2 i2x4;
```

Мы можем обращаться к элементам матрицы, используя синтаксис доступа к элементу массива по двум индексам. Например, для установки значения элемента (i, j) матрицы **M**, следует писать:

```
M[i][j] = value;
```

Кроме того, мы можем обращаться к элементам матрицы **M** как к членам структуры. Определены следующие имена элементов:

Если нумерация начинается с единицы:

```
M._11 = M._12 = M._13 = M._14 = 0.0f;
M._21 = M._22 = M._23 = M._24 = 0.0f;
M._31 = M._32 = M._33 = M._34 = 0.0f;
M._41 = M._42 = M._43 = M._44 = 0.0f;
```

Если нумерация начинается с нуля:

```
M._m00 = M._m01 = M._m02 = M._m03 = 0.0f;
M._m10 = M._m11 = M._m12 = M._m13 = 0.0f;
M._m20 = M._m21 = M._m22 = M._m23 = 0.0f;
M._m30 = M._m31 = M._m32 = M._m33 = 0.0f;
```

Иногда нам будет требоваться сослаться на отдельный вектор-строку матрицы. Это делается с использованием синтаксиса доступа к элементу массива по индексу. Например, чтобы получить *i*-ый вектор-строку матрицы **M**, следует написать:

```
vector ithRow = M[i]; // получить i-ый вектор-строку в M
```

ПРИМЕЧАНИЕ Для инициализации переменных в HLSL можно применять два варианта синтаксиса. Первый — синтаксис инициализации структуры:

```
vector u = {0.6f, 0.3f, 1.0f, 1.0f};
vector v = {1.0f, 5.0f, 0.2f, 1.0f};
```

Или эквивалентный стиль конструктора:

```
vector u = vector(0.6f, 0.3f, 1.0f, 1.0f);
vector v = vector(1.0f, 5.0f, 0.2f, 1.0f);
```

Вот еще несколько примеров:

```
float2x2 f2x2 = float2x2(1.0f, 2.0f, 3.0f, 4.0f);
int2x2 m = {1, 2, 3, 4};
int n = int(5);
int a = {5};
float3 x = float3(0, 0, 0);
```

16.3.4 Массивы

Мы можем объявить массив значений заданного типа используя синтаксис, аналогичный C++. Например:

```
float  M[4][4];
half   p[4];
vector v[12];
```

16.3.5 Структуры

Структуры объявляются точно так же, как это делается в C++. Однако, членами структур в HLSL не могут быть функции. Вот пример объявления структуры в HLSL:

```
struct MyStruct
{
     matrix T;
     vector n;
     float  f;
     int    x;
     bool   b;
};

MyStruct s; // создаем экземпляр
s.f = 5.0f; // доступ к члену
```

16.3.6 Ключевое слово typedef

Ключевое слово **typedef** делает в HLSL то же самое, что и в C++. Например, приведенный ниже фрагмент кода присваивает имя **point** типу **vector<float, 3>**:

```
typedef vector<float, 3> point;
```

Теперь вместо

```
vector<float, 3> myPoint;
```

мы можем писать

```
point myPoint;
```

Вот еще два примера, показывающие как можно использовать ключевое слово **typedef** с константными типами и массивами:

```
typedef const float CFLOAT;
typedef float point2[2];
```

16.3.7 Префиксы переменных

Приведенные ниже ключевые слова можно использовать в качестве префиксов при объявлении переменных:

- **`static`** — Если глобальная переменная объявляется с префиксом **`static`**, это означает, что переменная не должна быть доступна вне шейдера. Другими словами, она будет локальной для шейдера. Если же с префиксом **`static`** объявляется локальная переменная, то она будет вести себя так же, как локальная статическая переменная в C++. Это значит, что она инициализируется один раз при первом выполнении функции, а затем ее значение сохраняется между вызовами функции. Если в объявлении переменной нет инициализации, ей автоматически присваивается значение 0.

  ```
  static int x = 5;
  ```

- **`uniform`** — Если переменная объявлена с префиксом **`uniform`**, это означает, что она инициализируется вне шейдера, например в коде приложения, и передается в шейдер.
- **`extern`** — Если переменная объявлена с префиксом **`extern`**, это значит, что она должна быть доступна вне шейдера, например из кода приложения. Этот префикс можно указывать только для глобальных переменных. Нестатические глобальные переменные будут внешними по умолчанию.
- **`shared`** — Если переменная объявлена с префиксом **`shared`**, это указывает каркасу эффектов (см. главу 19), что переменная совместно используется несколькими эффектами. Префикс **`shared`** может использоваться только для глобальных переменных.
- **`volatile`** — Если переменная объявлена с префиксом **`volatile`**, это указывает каркасу эффектов (см. главу 19), что значение переменной будет часто изменяться. Префикс **`volatile`** может использоваться только для глобальных переменных.
- **`const`** — Ключевое слово **`const`** в HLSL имеет тот же самый смысл, что и в C++. Значит, если переменная объявлена с префиксом **`const`**, то она является константой и ее значение не может меняться.

  ```
  const float pi = 3.14f;
  ```

16.4 Ключевые слова, инструкции и приведение типов

16.4.1 Ключевые слова

Для справки мы приводим здесь ключевые слова, определенные в HLSL:

```
asm      bool     compile const       decl         do
double   else     extern  false       float        for
half     if       in      inline      inout        int
matrix   out      pass    pixelshader return       sampler
shared   static   string  struct      technique    texture
true     typedef  uniform vector      vertexshader void
volatile while
```

Во втором списке приведены идентификаторы, которые зарезервированы и в данный момент не используются, но в будущих версиях могут стать ключевыми словами:

```
auto             break    case    catch     char         class
const_cast       continue default delete    dynamic_cast enum
explicit         friend   goto    long      mutable      namespace
new              operator private protected public       register
reinterpret_cast short    signed  sizeof    static_cast  switch
template         this     throw   try       typename     union
unsigned         using    virtual
```

16.4.2 Поток выполнения программы

Набор поддерживаемых HLSL инструкций для ветвления, повторов и общего потока программы очень похож на инструкции C++. Синтаксис этих инструкций тоже полностью аналогичен C++.

Инструкция *Return*:

```
return (выражение);
```

Инструкции *If* и *If...Else*:

```
if(условие)
{
     инструкция(s);
}

if( условие )
{
     инструкция(s);
}
else
{
     инструкция(s);
}
```

Инструкция *for*:

```
for(инициализация; условие; изменение)
{
     инструкция(s);
}
```

Инструкция *while*:

```
while( условие )
{
     инструкция(s);
}
```

Инструкция *do...while*:

```
do
{
     инструкция(s);
} while(условие);
```

16.4.3 Приведение типов

HLSL поддерживает очень гибкую схему приведения типов. Синтаксис приведения в HLSL тот же самый, что и в C. Например, чтобы преобразовать значение типа **`float`** в значение типа **`matrix`**, мы напишем:

```
float f = 5.0f;
matrix m = (matrix)f;
```

В примерах из этой книги вы сможете понять смысл приведения из синтаксиса. Однако, если вам потребуется дополнительная информация по поддерживаемым приведениям типов, вы найдете ее в документации DirectX SDK, выбрав на вкладке Contents пункт DirectX Graphics\Reference\Shader Reference\High Level Shading Language\Type.

16.5 Операторы

HLSL поддерживает иного операторов, похожих на те, что есть в C++. За исключением нескольких особенностей, о которых мы поговорим ниже, эти операторы работают точно так же, как и их аналоги в C++. Ниже приведен список поддерживаемых в HLSL операторов:

```
[]      .       >       <       <=      >=

!=      ==      !       &&      ||      ?:

+       +=      -       -=      *       *=

/       /=      %       %=      ++      --

=       ()      ,
```

Хотя поведение операторов очень похоже на C++, есть несколько отличий. Во-первых, операция деления по модулю работает как для целых чисел, так и для чисел с плавающей точкой. Кроме того, для операции деления по модулю необходимо, чтобы у обоих ее операндов был один и тот же знак (то есть, чтобы либо оба операнда были положительными, либо оба отрицательными).

Во-вторых, обратите внимание, что большинство операторов HLSL действуют покомпонентно. Это вызвано тем фактом, что векторы и матрицы встроены в язык, а эти типы состоят из отдельных компонентов. Благодаря наличию операторов, работающих на уровне компонентов, такие операции как сложение векторов/матриц, вычитание векторов/матриц и проверка равенства векторов/матриц выполняются с использованием тех же операторов, которые применяются для скалярных типов. Взгляните на следующие примеры:

ПРИМЕЧАНИЕ Операторы ведут себя так, как ожидается для скаляров (то есть обычным для C++ образом).

```
vector u = {1.0f, 0.0f, -3.0f, 1.0f};
vector v = {-4.0f, 2.0f, 1.0f, 0.0f};

// Складываем соответствующие компоненты
vector sum = u + v; // сумма = (-3.0f, 2.0f, -2.0f, 1.0f)
```

Инкремент вектора увеличивает каждую из его компонент:

```
// До инкремента: sum = (-3.0f, 2.0f, -2.0f, 1.0f)

sum++; // После инкремента: sum = (-2.0f, 3.0f, -1.0f, 2.0f)
```

Покомпонентное произведение векторов:

```
vector u = {1.0f, 0.0f, -3.0f, 1.0f};
vector v = {-4.0f, 2.0f, 1.0f, 0.0f};

// Умножаем соответствующие компоненты
vector sum = u * v; // произведение = (-4.0f, 0.0f, -3.0f, 0.0f)
```

Операторы сравнения тоже работают покомпонентно и возвращают вектор или матрицу, каждый элемент которой является логическим значением. Полученный в результате «логический» вектор содержит результаты сравнения соответствующих компонент его операндов. Например:

```
vector u = { 1.0f, 0.0f, -3.0f, 1.0f};
vector v = {-4.0f, 0.0f, 1.0f, 1.0f};

vector b = (u == v); // b = (false, true, false, true)
```

И, в заключение, мы рассмотрим повышение типа переменной при бинарных операциях:

- Если в бинарной операции размер левого операнда отличается от размера правого операнда, то операнд меньшего размера повышается (приводится) до типа операнда большего размера. Например, если переменная **x** типа **float**, а переменная **y** типа **float3**, то в выражении **(x + y)** переменная **x** будет повышена до типа **float3** и результатом всего выражения также будет значение типа **float3**. При повышении типа

используются предопределенные правила приведения типов. В рассматриваемом случае мы преобразуем скаляр в вектор; следовательно, после повышения **x** до **float3**, **x = (x, x, x)**, как указано в правилах приведения скалярных типов к векторным. Помните, что результат повышения не определен, если не определена соответствующая операция приведения. Например, мы не можем выполнить повышение **float2** до **float3** поскольку такая операция приведения типа не существует.

- Если в бинарной операции диапазон значений левого операнда отличается от диапазона значений правого операнда, то операнд с меньшим диапазоном значений повышается (приводится) до типа операнда с большим диапазоном значений. Например, если переменная **x** типа **int**, а переменная **y** типа **half**, то в выражении **(x + y)** переменная **x** будет повышена до типа **half** и результатом всего выражения также будет значение типа **half**.

16.6 Определяемые пользователем функции

Функции в HLSL имеют следующие особенности:

- Синтаксис объявления функций такой же, как и в C++.
- Параметры всегда передаются по значению.
- Рекурсия не поддерживается.
- Функции всегда встраиваемые (inline).

Кроме того, в HLSL добавлено несколько дополнительных ключевых слов, которые могут использоваться в объявлениях функций. Для примера, рассмотрим приведенный ниже код функции на HLSL:

```
bool foo(in const bool b, // Входное значение bool
         out int r1,      // Выходное значение int
         inout float r2)  // Входное и выходное значение float
{
     if( b ) // Проверяем входное значение
     {
          r1 = 5; // Возвращаем значение через r1
     }
     else
     {
          r1 = 1; // Возвращаем значение через r1
     }
     // Поскольку r2 объявлена с ключевым словом inout
     // мы можем использовать ее как входное значение
     // и возвращать значения через нее
     r2 = r2 * r2 * r2;
     return true;
}
```

Функция почти полностью аналогична коду на C++, за исключением ключевых слов **in**, **out** и **inout**.

- **in** — Указывает что *аргумент* (конкретная переменная, которую мы передаем в параметре) должен быть скопирован в параметр перед началом выполнения функции. Не требуется явно указывать ключевое слово **in**, поскольку его наличие подразумевается по умолчанию. Например, следующие две записи эквивалентны:

```
float square(in float x)
{
     return x * x;
}
```

Без явного указания **in**:

```
float square(float x)
{
     return x * x;
}
```

- **out** — Указывает, что при возврате из функции параметр должен быть скопирован в аргумент. Это применяется для возврата значений через параметры. Ключевое слово **out** необходимо потому что HLSL не поддерживает передачу по ссылке или передачу указателя. Обратите внимание, что если параметр отмечен ключевым словом **out**, аргумент не копируется в параметр перед началом работы функции. Другими словами, такой параметр может использоваться только для возврата значений и не может применяться для передачи значений в функцию.

```
void square(in float x, out float y)
{
     y = x * x;
}
```

Здесь мы передаем возводимое в квадрат число через параметр **x**, а результат вычислений возвращаем через параметр **y**.

- **inout** — Данное сокращение означает, что параметр является как входным, так и выходным. Ключевое слово **inout** применяется в том случае, если вам надо использовать один и тот же параметр как для передачи значений в функцию, так и для возврата значений из нее.

```
void square(inout float x)
{
     x = x * x;
}
```

Здесь мы передаем возводимое в квадрат число через параметр **x** и через него же возвращаем вычисленное значение.

16.7 Встроенные функции

В HLSL есть богатый набор встроенных функций, часто используемых для трехмерной графики. Ниже приведена таблица с сокращенным списком этих функций. В следующих двух главах мы попрактикуемся в применении некоторых из них. А сейчас давайте просто познакомимся с этими функциями.

ПРИМЕЧАНИЕ Если в дальнейшем вам понадобится более подробная информация, полный список встроенных функций HLSL вы найдете в документации DirectX, перейдя на вкладку Contents и выбрав пункт DirectX Graphics\Reference\Shader Reference\High Level Shader Language\Intrinsic Functions.

Функция	Описание
`abs(x)`	Возвращает $\|x\|$.
`ceil(x)`	Возвращает наименьшее целое, которое больше или равно *x*.
`clamp(x, a, b)`	Ограничивает *x* в диапазоне [*a*, *b*] и возвращает результат. Если *x* меньше *a* возвращается *a*, если *x* больше *b*, возвращается *b*, в остальных случаях возвращается *x*.
`cos(x)`	Возвращает косинус *x*, *x* — в радианах.
`cross(u, v)`	Возвращает $\mathbf{u} \times \mathbf{v}$.
`degrees(x)`	Преобразует *x* из радиан в градусы.
`determinant(M)`	Возвращает детерминант матрицы *det*(**M**).
`distance(u, v)`	Возвращает расстояние $\|\mathbf{v} - \mathbf{u}\|$ между точкам **u** и **v**.
`dot(u, v)`	Возвращает $\mathbf{u} \cdot \mathbf{v}$.
`floor(x)`	Возвращает наибольшее целое, которое меньше или равно *x*.
`length(v)`	Возвращает $\|\mathbf{v}\|$.
`lerp(u, v, t)`	Линейная интерполяция между **u** и **v** на основании параметра *t*, находящегося в диапазоне [0, 1].
`log(x)`	Возвращает $\ln(x)$.
`log10(x)`	Возвращает $\log_{10}(x)$.
`log2(x)`	Возвращает $\log_2(x)$.
`max(x, y)`	Возвращает *x* если $x \geq y$, иначе возвращает *y*.
`min(x, y)`	Возвращает *x* если $x \leq y$, иначе возвращает *y*.
`mul(M, N)`	Возвращает произведение матриц **MN**. Обратите внимание, что произведение матриц **MN** должно быть определено. Если **M** это вектор, он используется как вектор-строка, если **N** это вектор, то он используется как вектор-столбец, чтобы было определено произведение матрицы на вектор.

Функция	Описание
normalize(v)	Возвращает **v**/|**v**|.
pow(b, n)	Возвращает b^n.
radians(x)	Преобразует *x* из градусов в радианы.
reflect(v, n)	Вычисляет вектор отражения по исходному вектору **v** и нормали поверхности **n**.
refract(v, n, eta)	Вычисляет вектор преломления по исходному вектору **v**, нормали поверхности **n** и отношению коэффициентов преломления двух материалов *eta*. Чтобы получить дополнительные сведения о преломлении, посмотрите закон Снеллиуса в учебнике физики.
rsqrt(x)	Возвращает $1/\sqrt{x}$.
saturate(x)	Возвращает **clamp(*x*, 0.0, 1.0)**.
sin(x)	Возвращает синус *x*, *x* в радианах.
sincos(in x, out s, out c)	Возвращает синус и косинус *x*, *x* в радианах.
sqrt(x)	Возвращает $\sqrt{x}$.
tan(x)	Возвращает тангенс *x*, *x* в радианах.
transpose(M)	Возвращает транспонированную матрицу $\mathbf{M}^T$.

Большинство функций имеют перегруженные версии для работы со всеми встроенными типами для которых результат работы функции имеет смысл. Например, функция **abs** имеет смысл для всех скалярных типов и, следовательно, имеет перегруженные версии для всех них. В то же время векторное произведение имеет смысл только для трехмерных векторов и поэтому для функции **cross** есть перегруженные версии для работы с трехмерными векторами любых типов (то есть векторами, компоненты которых имеют тип **int**, **float**, **double** и т.д.). С другой стороны, линейная интерполяция, **lerp**, имеет смысл для скаляров, двухмерных, трехмерных и четырехмерных векторов, и имеет перегруженные варианты для всех этих типов.

ПРИМЕЧАНИЕ Если вы передаете вектор или матрицу в «скалярную» функцию, то есть функцию, которая обычно выполняется над скалярами (например, **cos(x)**), вычисления будут проводиться покомпонентно. Например, если написать:

```
float3 v = float3(0.0f, 0.0f, 0.0f);
v = cos(v);
```

то функция будет применена к каждой компоненте : **v** = (cos(*x*), cos(*y*), cos(*z*)).

Приведенные ниже фрагменты кода демонстрируют использование встроенных функций:

```
float x = sin(1.0f);   // синус 1.0f радиан

float y = sqrt(4.0f);  // квадратный корень из 4.

vector u = {1.0f, 2.0f, -3.0f, 0.0f};
vector v = {3.0f, -1.0f, 0.0f, 2.0f};
float s = dot(u, v); // скалярное произведение векторов u и v.

float3 i = {1.0f, 0.0f, 0.0f};
float3 j = {0.0f, 1.0f, 0.0f};
float3 k = cross(i, j); // векторное произведение векторов i и j.

matrix<float, 2, 2> M = {1.0f, 2.0f, 3.0f, 4.0f};
matrix<float, 2, 2> T = transpose(M); // транспонирование матрицы
```

16.8 Итоги

- Мы пишем программы на HLSL в текстовом редакторе, сохраняем их в обычном текстовом файле ASCII, а затем компилируем их в нашем приложении с помощью функции **`D3DXCompileShaderFromFile`**.
- Интерфейс **`ID3DXConstantTable`** позволяет нам устанавливать значения переменных шейдеров из нашего приложения. Такое взаимодействие необходимо, поскольку значения используемых в шейдерах переменных могут меняться от кадра к кадру. Например, если в приложении изменяется матрица вида, нам необходимо обновить и переменную с матрицей вида в шейдере, записав в нее новые значения. Эти действия и позволяет выполнить интерфейс **`ID3DXConstantTable`**.
- Для каждого шейдера мы должны определить входную и выходную структуры, описывающие формат данных, которые шейдер получает и которые возвращает соответственно.
- У каждого шейдера есть функция, являющаяся точкой входа, которая в параметре получает входную структуру, использующуюся для передачи данных в шейдер. Кроме того, каждый шейдер возвращает экземпляр выходной структуры, использующийся для возврата данных из шейдера.

Глава 17
Знакомство с вершинными шейдерами

Вершинный шейдер (*vertex shader*) — это программа, выполняемая процессором видеокарты и заменяющая этапы преобразования и освещения в фиксированном конвейере. (Это описание не является абсолютно верным, поскольку если вершинные шейдеры не поддерживаются аппаратно, они могут эмулироваться программно библиотекой времени выполнения Direct3D.) На рис. 17.1 изображена та часть конвейера визуализации, которую заменяют вершинные шейдеры.

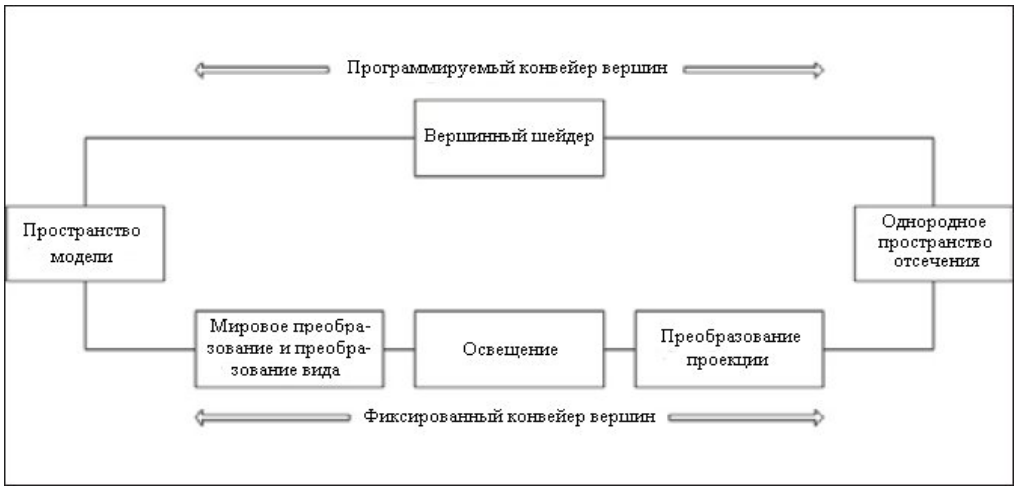

Рис. 17.1. *Вершинный шейдер заменяет этапы преобразования и освещения в фиксированном конвейере*

На рис. 17.1 видно, что на вход вершинного шейдера поступает вершина в локальной системе координат и шейдер должен вернуть освещенную (окрашенную) вершину в *однородном пространстве отсечения* (*homogeneous clip space*). (В этой книге, чтобы не усложнять материал, мы не будем углубляться в детали преобразования проекции. Отметим, что пространство, в которое матрица проекции преобразует вершины называется однородным пространством отсечения. Следовательно, для того, чтобы преобразовать вершину из локального пространства в однородное пространство отсечения, необходимо выполнить следующую последовательность преобразований: мировое преобразование,

преобразование вида и преобразование проекции выполняемые соответственно с помощью мировой матрицы, матрицы вида и матрицы проекции.) Для примитивов точек вершинные шейдеры могут использоваться также для управления размером.

Поскольку вершинные шейдеры — это программы, которые вы пишете сами (на HLSL) открывается *огромный* круг возможностей, позволяющих реализовать разнообразные графические эффекты. Например, благодаря вершинным шейдерам можно применять любой алгоритм освещения, реализуемый в коде шейдера. Мы больше не ограничены набором фиксированных алгоритмов освещения, реализованных в Direct3D. Более того, возможность управлять местоположением вершин также может применяться во многих задачах, таких как моделирование одежды, управление размером точек для систем частиц, смешивание вершин и морфинг. Помимо этого, в программируемом конвейере структура данных вершин более гибкая и может содержать больше данных, чем в фиксированном конвейере.

Вершинные шейдеры остаются достаточно новой возможностью и многие видеокарты не поддерживают их, особенно новые версии шейдеров, реализованные в DirectX 9. Чтобы проверить, какую версию шейдеров поддерживает установленная видеокарта, проверьте значение члена **VertexShaderVersion** структуры **D3DCAPS9** воспользовавшись макросом **D3DVS_VERSION**, как показано в приведенном ниже фрагменте кода:

```
// Если поддерживаемая устройством версия шейдеров меньше 2.0
if(caps.VertexShaderVersion < D3DVS_VERSION(2, 0))
      // Значит видеокарта не поддерживает шейдеры версии 2.0
```

Как видите, в двух параметрах **D3DVS_VERSION** передаются старший и младший номер версии соответственно. На данный момент функция **D3DXCompileShaderFromFile** поддерживает вершинные шейдеры версий 1.1 и 2.0.

Цели

- Изучить способы описания компонентов структуры данных вершины в программируемом конвейере.
- Узнать о различных способах использования компонентов вершины.
- Узнать как создать, установить и уничтожить вершинный шейдер.
- Изучить реализацию эффекта мультипликационной визуализации с помощью вершинных шейдеров.

17.1 Объявление вершин

До сих пор для описания компонентов нашей структуры данных вершины мы использовали настраиваемый формат вершин (FVF). Однако, в программируемом конвейере наши вершины могут содержать больше данных, чем можно описать с помощью FVF. Следовательно, нам нужен более наглядный и мощный способ объявления вершин.

ПРИМЕЧАНИЕ Мы можем продолжать использовать FVF для программируемого конвейера, если предоставляемых им возможностей достаточно для описания структур наших вершин. Но это лишь вопрос удобства, поскольку FVF автоматически преобразуется в объявление вершин.

17.1.1 Описание объявления вершин

Мы описываем объявление вершин в виде массива структур **D3DVERTEXELEMENT9**. Каждый элемент массива **D3DVERTEXELEMENT9** описывает один компонент данных вершины. Таким образом, если структура данных вершины содержит три компонента (например, местоположение, нормаль и цвет), соответствующее ей объявление вершины будет описано массивом из трех структур **D3DVERTEXELEMENT9**. Определение структуры **D3DVERTEXELEMENT9** выглядит следующим образом:

```
typedef struct _D3DVERTEXELEMENT9 {
     BYTE Stream;
     BYTE Offset;
     BYTE Type;
     BYTE Method;
     BYTE Usage;
     BYTE UsageIndex;
} D3DVERTEXELEMENT9;
```

- **Stream** — Указывает поток с которым связан данный компонент данных вершины.
- **Offset** — Смещение в байтах от начала структуры данных вершины до начала данных компонента. Например, если структура данных вершины объявлена следующим образом:

```
struct Vertex
{
     D3DXVECTOR3 pos;
     D3DXVECTOR3 normal;
};
```

Смещение компонента **pos** равно 0, поскольку этот компонент первый в структуре. Смещение компонента **normal** равно 12 потому что

sizeof(pos) == 12. Другими словами, компонент **normal** начинается с 12 байта, считая от начала структуры **Vertex**.

- **Type** — Указывает тип данных. Здесь можно использовать любой член из перечисления **D3DDECLTYPE**; чтобы посмотреть полный список типов, обратитесь к документации. Вот наиболее часто используемые типы:
  - **D3DDECLTYPE_FLOAT1** — Скаляр с плавающей точкой.
  - **D3DDECLTYPE_FLOAT2** — Двухмерный вектор с плавающей точкой.
  - **D3DDECLTYPE_FLOAT3** — Трехмерный вектор с плавающей точкой.
  - **D3DDECLTYPE_FLOAT4** — Четырехмерный вектор с плавающей точкой.
  - **D3DDECLTYPE_D3DCOLOR** — Значение типа **D3DCOLOR**, которое расширяется до цветового вектора RGBA с плавающей точкой (*r*, *g*, *b*, *a*), в котором каждая компонента нормализована в интервале [0, 1].
- **Method** — Задает мозаичный метод триангуляции. Мы считаем, что это достаточно сложная тема и поэтому будем всегда использоватть метод по умолчанию, задаваемый идентификатором **D3DDECLMETHOD_DEFAULT**.
- **Usage** — Указывает предполагаемый способ использования данного компонента. То есть позволяет определить, чем является данный компонент — вектором местоположения, вектором нормали, координатами текстуры и т.д. В качестве значений используются члены перечисления **D3DDECLUSAGE**:

```
typedef enum _D3DDECLUSAGE {
     D3DDECLUSAGE_POSITION     = 0,  // Местоположение
     D3DDECLUSAGE_BLENDWEIGHTS = 1,  // Веса смешивания
     D3DDECLUSAGE_BLENDINDICES = 2,  // Индексы смешивания
     D3DDECLUSAGE_NORMAL       = 3,  // Вектор нормали
     D3DDECLUSAGE_PSIZE        = 4,  // Размер точки
     D3DDECLUSAGE_TEXCOORD     = 5,  // Координаты текстуры
     D3DDECLUSAGE_TANGENT      = 6,  // Тангенциальный вектор
     D3DDECLUSAGE_BINORMAL     = 7,  // Бинормальный вектор
     D3DDECLUSAGE_TESSFACTOR   = 8,  // Мозаичный коэффициент
     D3DDECLUSAGE_POSITIONT    = 9,  // Преобразованная
                                     // позиция
     D3DDECLUSAGE_COLOR        = 10, // Цвет
     D3DDECLUSAGE_FOG          = 11, // Значение смешивания
                                     // тумана
     D3DDECLUSAGE_DEPTH        = 12, // Значение глубины
     D3DDECLUSAGE_SAMPLE       = 13  // Данные выборки
} D3DDECLUSAGE;
```

Тип **D3DDECLUSAGE_PSIZE** используется для задания размеров точек. Обычно он применяется для точечных спрайтов, чтобы можно было

управлять их размером для каждой вершины. Объявление вершины с флагом **D3DDECLUSAGE_POSITIONT** явно указывает, что эта вершина уже преобразована и, следовательно, процессор видеокарты не должен отправлять ее на этапы обработки вершин (преобразование и освещение).

ПРИМЕЧАНИЕ	Некоторые из типов использования, такие как **BLENDWEIGHTS**, **BLENDINDICES**, **TANGENT**, **BINORMAL** и **TESSFACTOR**, в этой книге не рассматриваются.

- **UsageIndex** — Используется для идентификации нескольких компонентов вершины с одинаковым типом использования. Индекс использования представляет собой целое число в диапазоне [0, 15]. Предположим, у нас есть три компонента вершины с флагом типа использования **D3DDECLUSAGE_NORMAL**. Тогда для первого из них мы должны указать индекс использования 0, для второго — индекс использования 1, и для третьего — индекс использования 2. Благодаря этому мы сможем идентифицировать отдельную нормаль по ее индексу использования.

Рассмотрим пример описания объявления вершины. Предположим, что описываемый формат содержит вектор местоположения и три вектора нормалей. В этом случае описание объявления вершины будет выглядеть так:

```
D3DVERTEXELEMENT9 decl[] =
{
     {0, 0,  D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
             D3DDECLUSAGE_POSITION, 0},
     {0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
             D3DDECLUSAGE_NORMAL, 0},
     {0, 24, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
             D3DDECLUSAGE_NORMAL, 1},
     {0, 36, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
             D3DDECLUSAGE_NORMAL, 2},
     D3DDECL_END()
};
```

Макрос **D3DDECL_END** применяется для инициализации последнего элемента в массиве **D3DVERTEXELEMENT9**. Кроме того, обратите внимание на применение индекса использования для отметки векторов нормалей.

17.1.2 Создание объявления вершин

Как только вы описали объявление вершины в массиве структур **D3DVERTEXELEMENT9**, можно получить указатель на интерфейс **IDirect3DVertexDeclaration9** с помощью следующего метода:

```
HRESULT IDirect3DDevice9::CreateVertexDeclaration(
     CONST D3DVERTEXELEMENT9* pVertexElements,
     IDirect3DVertexDeclaration9** ppDecl
);
```

- **pVertexElements** — Массив структур **D3DVERTEXELEMENT9**, описывающий объявление вершины, которое мы создаем.
- **ppDecl** — Используется для возврата указателя на созданный интерфейс **IDirect3DVertexDeclaration9**.

Вот пример вызова, где **decl** — это массив структур **D3DVERTEXELEMENT9**:

```
IDirect3DVertexDeclaration9* _decl = 0;
hr = _device->CreateVertexDeclaration(decl, &_decl);
```

17.1.3 Разрешение использования объявлений вершин

Вспомните, что настраиваемый формат вершин создан лишь для удобства и автоматически преобразуется в объявление вершин. Значит, когда мы используем объявление вершин непосредственно, нам больше не нужен вызов:

```
Device->SetFVF(fvf);
```

Вместо него мы используем вызов:

```
Device->SetVertexDeclaration(_decl);
```

где **_decl** — это указатель на интерфейс **IDirect3DVertexDeclaration9**.

17.2 Использование данных вершин

Рассмотрим следующее объявление вершины:

```
D3DVERTEXELEMENT9 decl[] =
{
     {0, 0,  D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
             D3DDECLUSAGE_POSITION, 0},
     {0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
             D3DDECLUSAGE_NORMAL, 0},
     {0, 24, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
             D3DDECLUSAGE_NORMAL, 1},
     {0, 36, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
             D3DDECLUSAGE_NORMAL, 2},
     D3DDECL_END()
};
```

Нам необходим способ указать соответствие между элементами объявления вершины и членами данных входной структуры вершинного шейдера. Это соответствие описывается во входной структуре путем указания для каждого члена данных конструкции вида **: *тип-использования*[*индекс-использования*]**. Эта запись идентифицирует элемент объявления вершины по его типу использования и индексу использования. Тот элемент данных вершины, который идентифицирован по указанным параметрам, будет отображен

на член данных входной структуры. Например, для приведенного выше примера определение входной структуры может выглядеть так:

```
struct VS_INPUT
{
     vector position    : POSITION;
     vector normal      : NORMAL0;
     vector faceNormal1 : NORMAL1;
     vector faceNormal2 : NORMAL2;
};
```

ПРИМЕЧАНИЕ Если мы не указываем индекс использования, подразумевается, что он равен 0. Таким образом запись **POSITION** означает то же самое, что и **POSITION0**.

Здесь элемент 0 в **decl**, идентифицируемый по типу использования **POSITION** и индексу использования 0, отображается на элемент входной структуры **position**. Элемент 1 в **decl**, идентифицируемый по типу использования **NORMAL** и индексу использования 0, отображается на элемент входной структуры **normal**. Элемент 2 в **decl**, идентифицируемый по типу использования **NORMAL** и индексу использования 1, отображается на элемент входной структуры **faceNormal1**. Элемент 3 в **decl**, идентифицируемый по типу использования **NORMAL** и индексу использования 2, отображается на элемент входной структуры **faceNormal2**.

Входная структура вершинного шейдера поддерживает следующие типы использования:

- **POSITION[*n*]** — Местоположение.
- **BLENDWEIGHTS[*n*]** — Веса смешивания.
- **BLENDINDICES[*n*]** — Индексы смешивания.
- **NORMAL[*n*]** — Вектор нормали.
- **PSIZE[*n*]** — Размер точки.
- **DIFFUSE[*n*]** — Рассеиваемый цвет.
- **SPECULAR[*n*]** — Отражаемый цвет.
- **TEXCOORD[*n*]** — Координаты текстуры.
- **TANGENT[*n*]** — Тангенциальный вектор.
- **BINORMAL[*n*]** — Бинормальный вектор.
- **TESSFACTOR[*n*]** — Мозаичный коэффициент.

Здесь *n* — это необязательное целое число в диапазоне [0, 15].

ПРИМЕЧАНИЕ Еще раз напоминаем, что некоторые типы использования, такие как **BLENDWEIGHTS**, **TANGENT**, **BINORMAL**, **BLENDINDICES** и **TESSFACTOR**, в этой книге не рассматриваются.

Кроме того, мы должны указать способ использования каждого члена данных и для выходной структуры. Должен ли член данных интерпретироваться как вектор местоположения, или как цвет, или как координаты текстуры? Видеокарта не знает, пока вы явно не укажете ей это. Синтаксис здесь тот же самый, что и во входной структуре:

```
struct VS_OUTPUT
{
     vector position : POSITION;
     vector diffuse  : COLOR0;
     vector specular : COLOR1;
};
```

Выходная структура вершинного шейдера поддерживает следующие типы использования:

- **POSITION** — Местоположение.
- **PSIZE** — Размер точки.
- **FOG** — Значение смешивания тумана.
- **COLOR[*n*]** — Цвет вершины. Обратите внимание, что можно возвращать несколько значений цветов. Для получения итогового цвета все эти цвета смешиваются вместе.
- **TEXCOORD[*n*]** — Координтаы текстуры. Обратите внимание, что можно возвращать несколько координат текстуры.

Здесь *n* — это необязательное целое число в диапазоне [0, 15].

17.3 Этапы работы с вершинным шейдером

В приведенном ниже списке перечислены этапы, необходимые для создания и использования вершинного шейдера.

1. Написать и скомпилировать вершинный шейдер.
2. Создать представляющий вершинный шейдер интерфейс **IDirect3DVertexShader9** на основе скомпилированного кода.
3. Установить вершинный шейдер с помощью метода **IDirect3DDevice9::SetVertexShader**.

И конечно же, мы должны уничтожить вершинный шейдер, когда работа с ним завершена. В следующих подразделах мы детально рассмотрим эти этапы.

17.3.1 Написание и компиляция вершинного шейдера

Сперва мы должны написать программу вершинного шейдера. В этой книге мы пишем наши шейдеры на HLSL. Когда код шейдера написан, мы компилируем его

с помощью функции **D3DXCompileShaderFromFile**, как было описано в разделе 16.2.2. Вспомните, что эта функция возвращает указатель на интерфейс **ID3DXBuffer**, который содержит скомпилированный код шейдера.

17.3.2 Создание вершинного шейдера

После того, как мы скомпилировали код шейдера, необходимо получить указатель на представляющий вершинный шейдер интерфейс **IDirect3DVertexShader9** с помощью следующего метода:

```
HRESULT IDirect3DDevice9::CreateVertexShader(
     const DWORD *pFunction,
     IDirect3DVertexShader9** ppShader
);
```

- **pFunction** — Указатель на скомпилированный код шейдера.
- **ppShader** — Возвращает указатель на интерфейс **IDirect3DVertexShader9**.

Предположим, переменная **shader** — это указатель на интерфейс **ID3DXBuffer**, содержащий скомпилированный код шейдера. Тогда для получения указателя на интерфейс **IDirect3DVertexShader9** следует написать:

```
IDirect3DVertexShader9* ToonShader = 0;
hr = Device->CreateVertexShader(
           (DWORD*)shader->GetBufferPointer(),
           &ToonShader);
```

ПРИМЕЧАНИЕ Повторим еще раз, **D3DXCompileShaderFromFile** — это функция, которая возвращает скомпилированный код шейдера (**shader**).

17.3.3 Установка вершинного шейдера

После того, как мы получили указатель на интерфейс **IDirect3DVertexShader9**, представляющий наш вершинный шейдер, мы должны разрешить его использование с помощью следующего метода:

```
HRESULT IDirect3DDevice9::SetVertexShader(
     IDirect3DVertexShader9* pShader
);
```

Метод получает единственный параметр в котором мы передаем указатель на тот вершинный шейдер, который должен быть включен. Чтобы включить шейдер, созданный в разделе 17.3.2, следует написать:

```
Device->SetVertexShader(ToonShader);
```

17.3.4 Уничтожение вершинного шейдера

Как и для всех интерфейсов Direct3D, здесь для очистки мы должны при завершении работы с интерфейсом вызвать его метод **`Release`**. Продолжая изучать пример шейдера созданного в разделе 17.3.2, мы получаем:

```
d3d::Release<IDirect3DVertexShader9*>(ToonShader);
```

17.4 Пример приложения: рассеянный свет

В качестве разминки напишем вершинный шейдер, который будет реализовать для вершин обычное рассеянное освещение для направленного (параллельного) источника света. Напомним, что для рассеянного света количество получаемого вершиной света вычисляется на основании угла между нормалью вершины и вектором света (который указывает в направлении на источник света). Чем меньше угол, тем больше света получает вершина, и чем больше угол, тем меньше света получает вершина. Если угол больше или равен 90 градусам, вершина вообще не освещена. Подробное описание алгоритма рассеянного освещения приводилось в разделе 13.4.1.

Начнем с исследования кода вершинного шейдера.

```
// Файл: diffuse.txt
// Описание: Вершинный шейдер, реализующий рассеянное освещение.

//
// Глобальные переменные используемые для хранения
// матрицы вида, матрицы проекции, фоновой составляющей материала,
// рассеиваемой составляющей материала и вектора освещения,
// указывающего в направлении на источник света.
// Все эти переменные инициализируются из приложения.
//

matrix ViewMatrix;
matrix ViewProjMatrix;

vector AmbientMtrl;
vector DiffuseMtrl;

vector LightDirection;

//
// Глобальные переменные, используемые для хранения
// интенсивности фонового света (фоновая составляющая
// испускаемого источником света) и интенсивности рассеиваемого
// света (рассеиваемая составляющая испускаемого источником света).
// Эти переменные инициализируются в шейдере.
//
vector DiffuseLightIntensity = {0.0f, 0.0f, 1.0f, 1.0f};
```

```
vector AmbientLightIntensity = {0.0f, 0.0f, 0.2f, 1.0f};

//
// Входная и выходная структуры
//
struct VS_INPUT
{
     vector position : POSITION;
     vector normal   : NORMAL;
};

struct VS_OUTPUT
{
     vector position : POSITION;
     vector diffuse  : COLOR;
};

//
// Точка входа
//
VS_OUTPUT Main(VS_INPUT input)
{
     // Обнуляем все члены экземпляра выходной структуры
     VS_OUTPUT output = (VS_OUTPUT)0;

     //
     // Преобразуем местоположение вершины в однородное пространство
     // отсечения и сохраняем его в члене output.position
     //
     output.position = mul(input.position, ViewProjMatrix);

     //
     // Преобразуем вектор освещения и нормаль в пространство вида.
     // Присваиваем компоненте w значение 0, поскольку мы преобразуем
     // векторы, а не точки.
     //
     LightDirection.w = 0.0f;
     input.normal.w   = 0.0f;
     LightDirection   = mul(LightDirection, ViewMatrix);
     input.normal     = mul(input.normal,   ViewMatrix);

     //
     // Вычисляем косинус угла между вектором света и нормалью
     //
     float s = dot(LightDirection, input.normal);

     //
     // Вспомните, что если угол между нормалью поверхности
     // и вектором освещения больше 90 градусов, поверхность
     // не получает света. Следовательно, если угол больше
     // 90 градусов, мы присваиваем s ноль, сообщая тем самым,
     // что поверхность не освещена.
     //
     if(s < 0.0f)
          s = 0.0f;
```

```
    //
    // Отраженный фоновый свет вычисляется путем покомпонентного
    // умножения вектора фоновой составляющей материала и вектора
    // интенсивности фонового света.
    //
    // Отраженный рассеиваемый свет вычисляется путем
    // покомпонентного умножения вектора рассеиваемой составляющей
    // материала на вектор интенсивности рассеиваемого света. Затем
    // мы масштабируем полученный вектор, умножая каждую его
    // компоненту на коэффициент затенения s, чтобы затемнить цвет
    // в зависимости от того, сколько света получает вершина
    // от источника.
    //
    // Сумма фоновой и рассеиваемой компонент дает нам
    // итоговый цвет вершины.
    //
    output.diffuse = (AmbientMtrl * AmbientLightIntensity) +
                        (s * (DiffuseLightIntensity * DiffuseMtrl));

    return output;
}
```

Теперь, когда мы посмотрели на код вершинного шейдера, давайте переключим передачу и взглянем на код приложения. В приложении используются следующие, относящиеся к рассматриваемой теме, глобальные переменные:

```
IDirect3DVertexShader9* DiffuseShader = 0;
ID3DXConstantTable* DiffuseConstTable = 0;

ID3DXMesh* Teapot               = 0;

D3DXHANDLE ViewMatrixHandle     = 0;
D3DXHANDLE ViewProjMatrixHandle = 0;
D3DXHANDLE AmbientMtrlHandle    = 0;
D3DXHANDLE DiffuseMtrlHandle    = 0;
D3DXHANDLE LightDirHandle       = 0;

D3DXMATRIX Proj;
```

У нас есть переменные, представляющие сам вершинный шейдер и его таблицу констант. Есть переменная для хранения сетки чайника, а за ней идет набор переменных **D3DXHANDLE**, чьи имена показывают для связи с какими переменными шейдера они используются.

Функция **Setup** выполняет следующие действия:

- Создает сетку чайника.
- Компилирует вершинный шейдер.
- Создает вершинный шейдер на основе скомпилированного кода.
- Получает через таблицу констант дескрипторы нескольких переменных программы шейдера.
- Инициализирует через таблицу констант некоторые переменные шейдера.

ПРИМЕЧАНИЕ В данном приложении для структуры данных вершины не требуются никакие дополнительные компоненты, которые нельзя описать с помощью настраиваемого формата вершин. Поэтому в данном примере мы используем настраиваемый формат вершин, а не объявление вершин. Вспомните, что описание настраиваемого формата вершин автоматически преобразуется в объявление вершин.

```
bool Setup()
{
     HRESULT hr = 0;

     //
     // Создание геометрии:
     //

     D3DXCreateTeapot(Device, &Teapot, 0);

     //
     // Компиляция шейдера
     //

     ID3DXBuffer* shader      = 0;
     ID3DXBuffer* errorBuffer = 0;

     hr = D3DXCompileShaderFromFile(
               "diffuse.txt",
               0,
               0,
               "Main", // имя точки входа
               "vs_1_1",
               D3DXSHADER_DEBUG,
               &shader,
               &errorBuffer,
               &DiffuseConstTable);

     // Выводим сообщения об ошибках
     if(errorBuffer)
     {
          ::MessageBox(0, (char*)errorBuffer->GetBufferPointer(),
                       0, 0);
          d3d::Release<ID3DXBuffer*>(errorBuffer);
     }

     if(FAILED(hr))
     {
          ::MessageBox(0, "D3DXCompileShaderFromFile() - FAILED",
                       0, 0);
          return false;
     }

     //
     // Создаем шейдер
     //

     hr = Device->CreateVertexShader(
             (DWORD*)shader->GetBufferPointer(),
             &DiffuseShader);
```

```
    if(FAILED(hr))
    {
        ::MessageBox(0, "CreateVertexShader - FAILED", 0, 0);
        return false;
    }

    d3d::Release<ID3DXBuffer*>(shader);

    //
    // Получаем дескрипторы
    //
    ViewMatrixHandle     = DiffuseConstTable->GetConstantByName(
                                                  0, "ViewMatrix");
    ViewProjMatrixHandle = DiffuseConstTable->GetConstantByName(
                                              0, "ViewProjMatrix");
    AmbientMtrlHandle    = DiffuseConstTable->GetConstantByName(
                                                 0, "AmbientMtrl");
    DiffuseMtrlHandle    = DiffuseConstTable->GetConstantByName(
                                                 0, "DiffuseMtrl");
    LightDirHandle       = DiffuseConstTable->GetConstantByName(
                                              0, "LightDirection");

    //
    // Устанавливаем константы шейдера:
    //

    // Направление на источник света:
    D3DXVECTOR4 directionToLight(-0.57f, 0.57f, -0.57f, 0.0f);
    DiffuseConstTable->SetVector(Device, LightDirHandle,
                                        &directionToLight);

    // Материалы:
    D3DXVECTOR4 ambientMtrl(0.0f, 0.0f, 1.0f, 1.0f);
    D3DXVECTOR4 diffuseMtrl(0.0f, 0.0f, 1.0f, 1.0f);
    DiffuseConstTable->SetVector(Device, AmbientMtrlHandle,
                                 &ambientMtrl);
    DiffuseConstTable->SetVector(Device, DiffuseMtrlHandle,
                                 &diffuseMtrl);
    DiffuseConstTable->SetDefaults(Device);

    // Вычисляем матрицу проекции
    D3DXMatrixPerspectiveFovLH(
        &Proj, D3DX_PI * 0.25f,
        (float)Width / (float)Height, 1.0f, 1000.0f);

    return true;
}
```

Функция **`Display`** достаточно простая. Она проверяет какие клавиши нажал пользователь и соотвествующим образом изменяет матрицу вида. Однако, поскольку преобразование вида мы выполняем в шейдере, нам необходимо обновить значение переменной шейдера, которая хранит матрицу вида. Мы делаем это через таблицу констант:

```
bool Display(float timeDelta)
{
    if(Device)
```

```
    {
        //
        // Код обновления матрицы вида пропущен...
        //

        D3DXMATRIX V;
        D3DXMatrixLookAtLH(&V, &position, &target, &up);

        DiffuseConstTable->SetMatrix(Device, ViewMatrixHandle, &V);

        D3DXMATRIX ViewProj = V * Proj;
        DiffuseConstTable->SetMatrix(Device, ViewProjMatrixHandle,
                                     &ViewProj);

        //
        // Визуализация
        //

        Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                      0xffffffff, 1.0f, 0);
        Device->BeginScene();

        Device->SetVertexShader(DiffuseShader);

        Teapot->DrawSubset(0);

        Device->EndScene();
        Device->Present(0, 0, 0, 0);
    }
    return true;
}
```

Обратите внимание, что мы включаем вершинный шейдер, который хотим использовать, перед вызовом **`DrawSubset`**.

Очистка выполняется как обычно; мы просто освобождаем все запрошенные интерфейсы:

```
void Cleanup()
{
    d3d::Release<ID3DXMesh*>(Teapot);
    d3d::Release<IDirect3DVertexShader9*>(DiffuseShader);
    d3d::Release<ID3DXConstantTable*>(DiffuseConstTable);
}
```

17.5 Пример приложения: мультипликационная визуализация

В качестве второго примера давайте напишем два вершинных шейдера, которые будут выполнять затенение и обвод контуров рисунка в мультипликационном стиле (рис. 17.2).

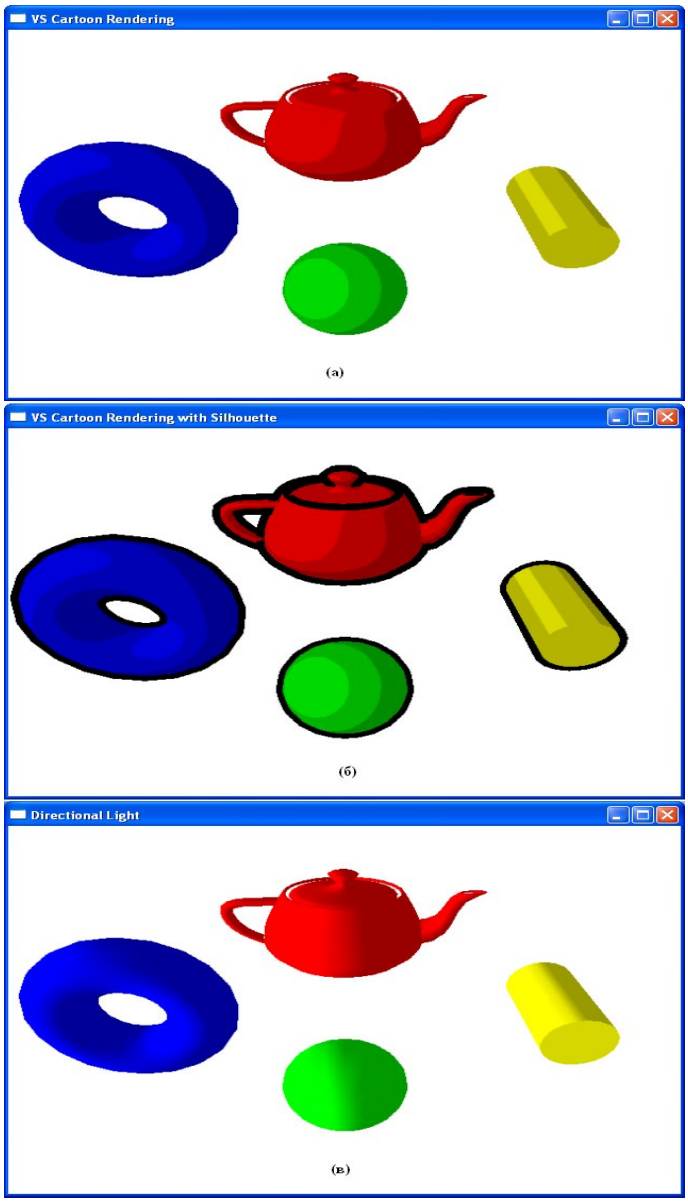

Рис. 17.2. *(а) Объекты с затенением выполненным по мультипликационной технологии (обратите внимание на резкие переходы между оттенками) (б) Усиление эффекта достигается путем обведения силуэта объекта (в) Объекты, затеняемые с использованием стандартного рассеянного освещения*

ПРИМЕЧАНИЕ *Мультипликационная визуализация* (*cartoon rendering*) — это один из видов *нефотореалистичной визуализации* (*non-photorealistic rendering*) иногда называемой *стилистической визуализацией* (*stylistic rendering*).

Мультипликационная визуализация подходит не для всех игр, например она будет лишней в реалистичной игре с видом от первого лица. С другой стороны, она может значительно улучшить атмосферу тех игр, где надо передать ощущения, возникающие при просмотре мультфильма. Кроме того, мультипликационная визуализация достаточно проста в реализации и замечательно подходит для демонстрации возможностей вершинных шейдеров.

Мы разделим мультипликационную визуализацию на два этапа.

1. Мультипликационные рисунки обычно имеют несколько уровней интенсивности затенения с резкими переходами от одного уровня к другому; мы будем ссылаться на такой способ как на *мультипликационное затенение* (*cartoon shading*). На рис. 17.2(а) видно, что для затенения сеток используются всего три уровня интенсивности (яркий, средний и темный) и переходы между оттенками явно выражены в отличие от рис. 17.2(в), где показан плавный переход от темного оттенка к светлому.
2. Также в мультфильмах обычно обводится силуэт объектов, как показано на рис. 17.2(б).

Оба этапа требуют собственных вершинных шейдеров.

17.5.1 Мультипликационное затенение

Для реализации мультипликационного затенения мы воспользуемся методикой, описанной Ландером в статье «Shades of Disney: Opaquing a 3D World», опубликованной в выпуске журнала Game Developer Magazine за март 2000 года. Работает она следующим образом: мы создаем состоящую из оттенков серого текстуру, которая будет управлять яркостью и должна состоять из требуемого нам количества оттенков. На рис. 17.3 показана текстура, которая будет использоваться в данном примере.

Рис. 17.3. *Текстура затенения содержит используемые градации яркости. Обратите внимание на резкие переходы между оттенками и на то, что яркость оттенков увеличивается слева направо*

Затем в вершинном шейдере мы выполняем стандартные для рассеянного освещения вычисления, определяя с помощью скалярного произведения косинус угла между нормалью вершины $\hat{\mathbf{N}}$ и вектором света $\hat{\mathbf{L}}$, который используется для определения того, сколько света получает вершина:

$$\hat{\mathbf{L}} \cdot \hat{\mathbf{N}} = s$$

Если $s < 0$, это значит, что угол между вектором света и нормалью вершины больше 90 градусов и, следовательно, поверхность не освещена. Поэтому, если $s < 0$, мы считаем что $s = 0$ и получаем значение s находящееся в диапазоне [0, 1].

Теперь в обычной модели рассеянного освещения мы использовали s для масштабирования цветового вектора, чтобы цвет становился темнее в зависимости от количества получаемого поверхностью света:

$$diffuseColor = s\,(r,\quad g,\quad b,\quad a)$$

Однако, в результате мы получаем плавные переходы от светлых оттенков к темным. Это не то, что нам надо для мультипликационного затенения. Нам нужны четко выраженные переходы между несколькими оттенками (для мультипликационной визуализации хорошо подходит использование от двух до четырех оттенков).

Вместо того, чтобы использовать s для масштабирования цветового вектора, мы будем использовать его как координату текстуры u для текстуры затенения, о которой мы говорили раньше и показали на рис. 17.3.

ПРИМЕЧАНИЕ Скаляр s является корректной координатой текстуры, поскольку s находится в диапазоне [0, 1], а это стандартный интервал координат текстуры.

В результате затенение вершин будет не плавным а резким. Например, текстура яркости может быть разделена на три оттенка, как показано на рис. 17.4.

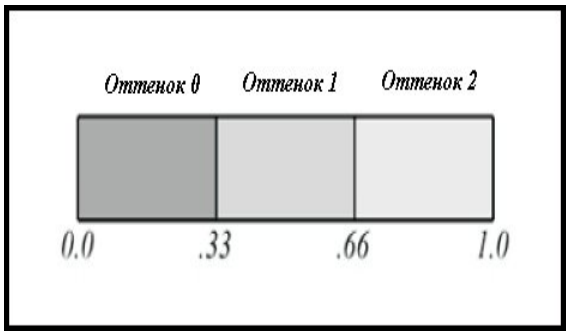

Рис. 17.4. *Используемый оттенок зависит от интервала, в который попадает координата текстуры*

Тогда если значение s находится в диапазоне [0, 0.33] для затенения используется *оттенок 0*, если значение s находится в диапазоне (0.33, 0.66] — используется *оттенок 1*, и для значений s из диапазона (0.66, 1] используется *оттенок 2*. Естественно, переходы от одного оттенка к другому будут резкими, что нам и требуется.

ПРИМЕЧАНИЕ Для мультипликационного затенения мы выключаем фильтрацию текстур, поскольку использование фильтрации сглаживает переходы между оттенками. Этот эффект нежелателен, поскольку нам требуются резкие переходы.

17.5.2 Код вершинного шейдера для мультипликационного затенения

Теперь мы представляем вершинный шейдер для мультипликационного затенения. Главной задачей шейдера является простое вычисление коэффициента $s = \hat{L} \cdot \hat{N}$ и установка соответствующих координат текстуры. Обратите внимание, что в выходную структуру мы добавили член данных для хранения вычисленных координат текстуры. Также заметьте, что мы по прежнему возвращаем цвет вершины, хотя не модифицируем его, а эффект затенения возникает, когда цвет вершины комбинируется с текстурой яркости.

```
// Файл:     toon.txt
// Описание: Вершинный шейдер, выполняющий освещение объектов
//           с созданием мультипликационного эффекта.

//
// Глобальные переменные
//

extern matrix WorldViewMatrix;
extern matrix WorldViewProjMatrix;

extern vector Color;

extern vector LightDirection;

static vector Black = {0.0f, 0.0f, 0.0f, 0.0f};

//
// Структуры
//

struct VS_INPUT
{
     vector position : POSITION;
     vector normal   : NORMAL;
};

struct VS_OUTPUT
{
     vector position : POSITION;
     float2 uvCoords : TEXCOORD;
     vector diffuse  : COLOR;
};

//
// Точка входа
//

VS_OUTPUT Main(VS_INPUT input)
{
```

```
    // Обнуляем члены выходной структуры
    VS_OUTPUT output = (VS_OUTPUT)0;

    // Преобразуем местположение вершины в однородное
    // пространство отсечения
    output.position = mul(input.position, WorldViewProjMatrix);

    //
    // Преобразуем вектор света и нормаль в пространство вида.
    // Мы устанавливаем значение компоненты w равным нулю потому что
    // мы преобразуем векторы. Предполагается, что в мировой матрице
    // нет никакого масштабирования.
    //
    LightDirection.w = 0.0f;
    input.normal.w   = 0.0f;
    LightDirection   = mul(LightDirection, WorldViewMatrix);
    input.normal     = mul(input.normal, WorldViewMatrix);

    //
    // Вычисление одномерной координаты текстуры
    // для мультипликационной визуализации
    //
    float u = dot(LightDirection, input.normal);

    //
    // Если u меньше нуля, считаем, что u равно нулю, поскольку
    // отрицательные значения соответствуют углам между вектором
    // света и нормалью большим 90 градусов. А если угол больше
    // 90 градусов, значит поверхность не освещается
    //
    if(u < 0.0f)
        u = 0.0f;

    //
    // Устанавливаем вторую координату текстуры на середину
    //
    float v = 0.5f;
    output.uvCoords.x = u;
    output.uvCoords.y = v;

    // Сохраняем цвет
    output.diffuse = Color;

    return output;
}
```

И несколько комментариев:

- Мы предполагаем, что в мировой матрице нет операций масштабирования поскольку в ином случае может измениться длинна и направление перемножаемых векторов.
- Мы всегда устанавливаем координату текстуры *v* на середину текстуры. Это подразумевает, что мы используем только одну горизонтальную линию текстуры, а значит можем использовать одномерную текстуру яркости вместо двухмерной. Впрочем, программа нормально работает как с одномерными так и с двухмерными текстурами. В примере мы используем двухмерную текстуру, а не одномерную, но для этого нет никаких веских причин.

17.5.3 Обводка силуэта

Чтобы усилить мультипликационный эффект, нам надо обвести силуэты объектов. Сделать это несколько сложнее, чем реализовать мультипликационное затенение.

17.5.3.1. Представление краев

Мы представляем край сетки в виде квадрата, образованного из двух треугольников (рис. 17.5).

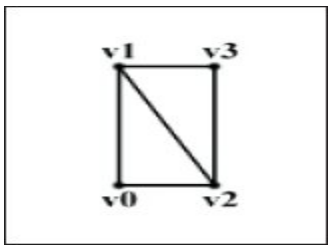

Рис. 17.5. *Квадрат, представляющий край*

Мы выбрали квадрат по двум причинам: во-первых можно легко изменять толщину края просто меняя размеры квадрата, и во-вторых мы можем визуализировать вырожденные квадраты для скрытия отдельных краев, например, тех краев, которые *не* являются частью силуэта. В Direct3D мы создаем квадрат из двух треугольников. *Вырожденный квадрат* (*degenerate quad*) — это квадрат, созданный из двух вырожденных треугольников. *Вырожденный треугольник* (*degenerate triangle*) — это треугольник с нулевой площадью или, другими словами, треугольник у которого все три вершины лежат на одной прямой. Если передать вырожденный треугольник в конвейер визуализации, то ничего отображено не будет. Это очень полезно, поскольку если мы хотим скрыть какой-нибудь треугольник, достаточно просто сделать его вырожденным без действительного удаления из списка треугольников (буфера вершин). Вспомните, что нам надо отображать только края силуэта, а не все края сетки.

Когда мы впервые создаем край, то указываем его четыре вершины таким образом, чтобы квадрат был вырожденный (рис. 17.6), а это значит, что данный край будет скрыт (не будет отображаться при визуализации).

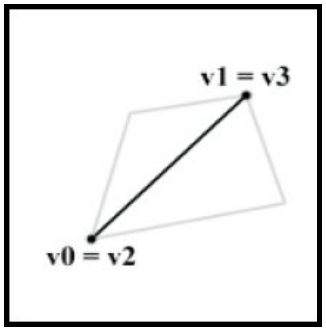

Рис. 17.6. *Вырожденный квадрат, описывающий край, разделенный на два треугольника*

Обратите внимание, что для двух вершин, $\mathbf{v}_0$ и $\mathbf{v}_1$ на рис. 17.6, мы указываем в качестве вектора нормали вершины нулевой вектор. Затем, когда мы передаем

вершины края в вершинный шейдер, он проверяет, является ли данный край частью силуэта. Если да, вершинный шейдер смещает позицию вершин вдоль вектора нормали вершины на заданный скаляр. Обратите внимание, что те вершины, для которых указан нулевой вектор нормали, не смещаются. Таким образом мы получаем невырожденный квадрат, представляющий край силуэта, как показано на рис. 17.7.

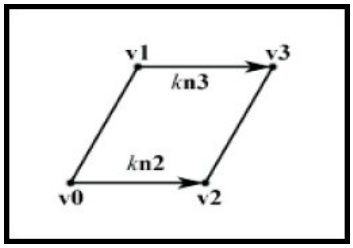

Рис. 17.7. *Вершины v_2 и v_3 края силуэта смещаются в направлении их нормалей вершин n_2 и n_3 соответственно. Обратите внимание, что вершины v_0 и v_1 остаются на фиксированных позициях и никуда не смещаются, поскольку их векторы нормалей вершин — нулевые векторы. Благодаря этому происходит успешное восстановление квадрата, представляющего край силуэта*

ПРИМЕЧАНИЕ Если векторы нормалей для вершин $\mathbf{v}_0$ и $\mathbf{v}_1$ будут ненулевыми, то эти вершины также будут смещаться. Но если мы сместим все четыре вершины, описывающие край силуэта, то просто переместим вырожденный квадрат. Зафиксировав вершины $\mathbf{v}_0$ и $\mathbf{v}_1$ и смещая только вершины $\mathbf{v}_2$ и $\mathbf{v}_3$ мы восстанавливаем квадрат.

17.5.3.2. Проверка для краев силуэта

Край является частью силуэта если он находится на стыке двух граней $face_0$ и $face_1$, и эти грани ориентированы в различных направлениях относительно вектора взгляда. То есть, если одна из граней является фронтальной, а другая — обратной, то край между ними является частью силуэта. На рис. 17.8 приведены примеры краев являющихся и не являющихся частью силуэта.

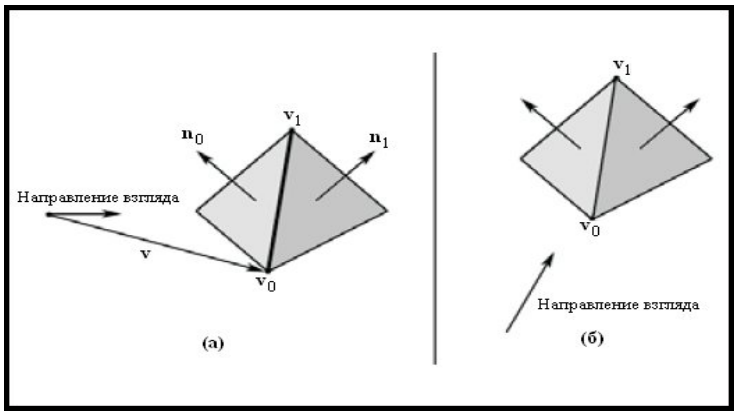

Рис. 17.8. *На рис. (а) одна из граней, совместно использующих край, образованный вершинами v_0 и v_1 является фронтальной, а другая — обратной, следовательно край является частью силуэта. На рис. (б) обе грани, совместно использующие край, образованный вершинами v_0 и v_1 являются фронтальными и, следовательно, край не является частью силуэта*

Из вышеприведенного материала следует, что для того, чтобы определить является ли вершина частью силуэта, нам надо знать векторы нормалей граней $face_0$ и $face_1$, к которым относится данная вершина. Это отражено в структуре данных вершины края:

```
struct VS_INPUT
{
     vector position    : POSITION;
     vector normal      : NORMAL0;
     vector faceNormal1 : NORMAL1;
     vector faceNormal2 : NORMAL2;
};
```

Первые два компонента мы уже обсуждали, но сейчас настало время взглянуть на два дополнительных вектора нормали — **`faceNormal1`** и **`faceNormal2`**. Эти векторы являются нормалями тех двух граней на стыке которых находится рассматриваемый край, а именно $face_0$ и $face_1$.

Математическая часть проверки, является ли вершина частью силуэта, заключается в следующем. Предположим, мы находимся в пространстве вида. Пусть $\mathbf{v}$ — это вектор, направленный от начала координат до проверяемой вершины (рис. 17.8). Пусть $\mathbf{n}_0$ — это нормаль грани $face_0$, а $\mathbf{n}_1$ — нормаль грани $face_1$. Тогда вершина является частью силуэта, если следующее сравнение истино:

$$(1) \quad (\mathbf{v}\cdot\mathbf{n}_0)(\mathbf{v}\cdot\mathbf{n}_1) < 0$$

Выражение (1) истино если знаки результатов двух скалярных произведений различны, что делает левую часть формулы отрицательной. Вспомните свойства скалярного произведения — если знаки двух скалярных произведений различны, это значит, что одна грань фронтальная, а другая — обратная.

Теперь рассмотрим случай, когда край образован только одной гранью, как показано на рис. 17.9, чья нормаль хранится в **`faceNormal1`**.

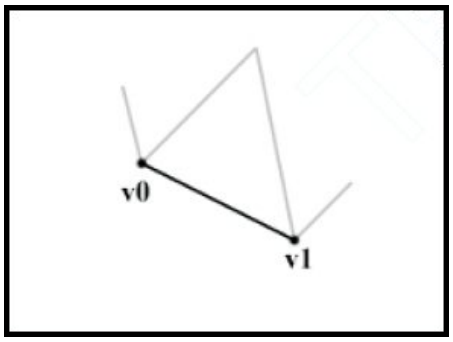

Рис. 17.9. *Край, определенный вершинами $\mathbf{v}_0$ и $\mathbf{v}_1$ используется только одной гранью*

Мы считаем, что такие края *всегда* являются частью силуэта. Чтобы вершинный шейдер обрабатывал такие грани как часть силуэта, мы устанавливаем что **`faceNormal2 = -faceNormal1`**. таким образом, нормали граней будут направлены в разные стороны и формула (1) будет истинна, указывая, что край является частью силуэта.

17.5.3.3. Генерация краев

Генерация краев сетки очень проста; мы перебираем грани сетки и для каждой стороны грани формируем квадрат (вырожденный, как показано на рис. 17.6).

ПРИМЕЧАНИЕ У каждой грани три стороны, так что для каждого треугольника формируется три края.

Для вершин каждого края нам также необходимо знать две грани между которыми этот край находится. Одна из граней — это текущий треугольник. Например, если мы вычисляем край *i*-ой грани, то *i*-ая грань участвует в формировании края. Другую участвующую в формировании края грань можно найти с помощью данных о смежности граней сетки.

17.5.4 Код вершинного шейдера обводки силуэта

Настала пора представить код вершинного шейдера для визуализации силуэта. Основная задача шейдера заключается в определении того, является ли переданная вершина частью силуэта. Если да, то вершинный шейдер осуществляет сдвиг вершины на заданный скаляр в направлении вектора нормали вершины.

```
// Файл:     outline.txt
// Описание: Вершинный шейдер, отображающий силуэт

//
// Глобальные переменные
//

extern matrix WorldViewMatrix;
extern matrix ProjMatrix;

static vector Black = {0.0f, 0.0f, 0.0f, 0.0f};

//
// Структуры
//

struct VS_INPUT
{
     vector position    : POSITION;
     vector normal      : NORMAL0;
     vector faceNormal1 : NORMAL1;
     vector faceNormal2 : NORMAL2;
};

struct VS_OUTPUT
{
     vector position : POSITION;
     vector diffuse  : COLOR;
};
```

```
//
// Точка входа
//

VS_OUTPUT Main(VS_INPUT input)
{
     // Обнуляем члены выходной структуры
     VS_OUTPUT output = (VS_OUTPUT)0;

     // Преобразуем местоположение в пространство вида
     input.position = mul(input.position, WorldViewMatrix);

     // Вычисляем вектор направления взгляда на вершину.
     // Вспомните, что в пространстве вида зритель находится
     // в начале координат (зритель это то же самое, что и камера).
     vector eyeToVertex = input.position;

     // Преобразуем нормали в пространство вида. Компоненте w
     // присваиваем нуль, поскольку преобразуем векторы.
     // Предполагается, что в мировой матрице нет масштабирования
     input.normal.w      = 0.0f;
     input.faceNormal1.w = 0.0f;
     input.faceNormal2.w = 0.0f;

     input.normal      = mul(input.normal, WorldViewMatrix);
     input.faceNormal1 = mul(input.faceNormal1, WorldViewMatrix);
     input.faceNormal2 = mul(input.faceNormal2, WorldViewMatrix);

     // Вычисляем косинус угла между вектором eyeToVertex
     // и нормалями граней
     float dot0 = dot(eyeToVertex, input.faceNormal1);
     float dot1 = dot(eyeToVertex, input.faceNormal2);

     // Если у косинусов разные знаки (один положительный,
     // а другой отрицательный) значит край является частью
     // силуэта
     if((dot0 * dot1) < 0.0f)
     {
          // Знаки разные, значит вершина является частью
          // края силуэта, смещаем позиции вершин на заданный
          // скаляр в направлении нормали вершины
          input.position += 0.1f * input.normal;
     }

     // Преобразование в однородное пространство отсечения
     output.position = mul(input.position, ProjMatrix);

     // Устанавливаем цвет силуэта
     output.diffuse = Black;

     return output;
}
```

17.6 Итоги

- Вершинные шейдеры позволяют заменять этапы преобразования и освещения в фиксированном конвейере функций. Заменяя фиксированный процесс на нашу собственную программу (вершинный шейдер) мы получаем *огромное* количество возможностей для реализации различных графических эффектов.
- Объявление вершин используется для описания формата данных наших вершин. Оно похоже на настраиваемый формат вершин (FVF), но является более гибким и позволяет описывать форматы вершин, которые не могут быть представлены посредством FVF. Обратите внимание, что если наша вершина может быть представлена с помощью FVF, мы можем продолжать использовать этот формат; внутри программы он будет автоматически преобразован в объявление вершин.
- Для входных данных семантика использования определяет каким образом будут связаны компоненты вершины из объявления вершин и переменные в программе на HLSL. Для выходных данных семантика использования определяет для чего будет использоваться компонент вершины (т.е. для указания местоположения, цвета, координат текстуры и т.д.).

Глава 18
Знакомство с пиксельными шейдерами

Пиксельный шейдер (*pixel shader*) — это программа, выполняемая процессором видеокарты во время процесса растеризации для каждого пикселя. (В отличие от вершинных шейдеров, Direct3D не может программно эмулировать пиксельные шейдеры.) Пиксельные шейдеры заменяют этап мультитекстурирования в фиксированном конвейере функций и предоставляют нам возможность непосредственно управлять отдельными пикселями и получить доступ к текстурным координатам для каждого пикселя. Прямой доступ к пикселям и координатам текстуры позволяет нам реализовать множество специальных эффектов, таких как мультитекстурирование, попиксельное освещение, глубина резкости, моделирование облаков, имитация огня и сложные способы затенения.

Вы можете проверить, какую версию пиксельных шейдеров использует установленная видеокарта, с помощью члена **`PixelShaderVersion`** структуры **`D3DCAPS9`** и макроса **`D3DPS_VERSION`**. Эта проверка показана в приведенном ниже фрагменте кода:

```
// Если поддерживаемая устройством версия меньше 2.0
if(caps.PixelShaderVersion < D3DPS_VERSION(2, 0))
   // Значит пиксельные шейдеры версии 2.0 устройство не поддерживает
```

Цели

- Получить базовое представление о концепции мультитекстурирования.
- Узнать как написать, создать и использовать пиксельный шейдер.
- Узнать реализовать мультитекстурирование с помощью пиксельного шейдера.

18.1 Основы мультитекстурирования

Мультитекстурирование, возможно, является самой простой из техник, которые можно реализовать с помощью пиксельного шейдера. Более того, поскольку пиксельные шейдеры заменяют стадию мультитекстурирования фиксированного конвейера, нам необходимо иметь хотя бы базовое представление о том, что происходит на этапе мультитекстурирования. Данный раздел представляет собой краткий обзор мультитекстурирования.

Когда ранее, в главе 6, мы обсуждали наложение текстур, обсуждение мультитекстурирования в фиксированном конвейере функций было пропущено по двум причинам. Во-первых, мультитекстурирование это часть более сложного процесса и мы посчитали, что в тот момент эта тема была бы слишком сложна для восприятия. Во-вторых, фиксированные функции этапа мультитекстурирования заменяются новыми и более мощными пиксельными шейдерами; следовательно имеет смысл не тратить время на устаревшие фиксированные функции этапа мультитекстурирования.

Идея, лежащая в основе мультитекстурирования отчасти связана со смешиванием. В главе 7 мы узнали о смешивании растеризуемых пикселей с пикселями ранее записанными во вторичный буфер для реализации некоторых эффектов. Теперь мы расширим эту идею для нескольких текстур. Итак, мы разрешаем одновременное использование нескольких текстур и затем определяем, как эти текстуры должны быть смешаны одна с другой для получения требуемого эффекта. Наиболее часто мультитекстурирование используется для освещения. Вместо того, чтобы воспользоваться моделью освещения Direct3D на этапе обработки вершин, мы применяем специальные карты текстур, называемые *картами освещения* (*light map*), которые сообщают о том, как освещена поверхность. Представим, например, что нам надо осветить прожектором большой ящик. Мы можем описать прожектор в виде структуры **`D3DLIGHT9`**, или можно смешать вместе карту текстур, представляющую ящик и карту освещения, представляющую прожектор, как показано на рис 18.1.

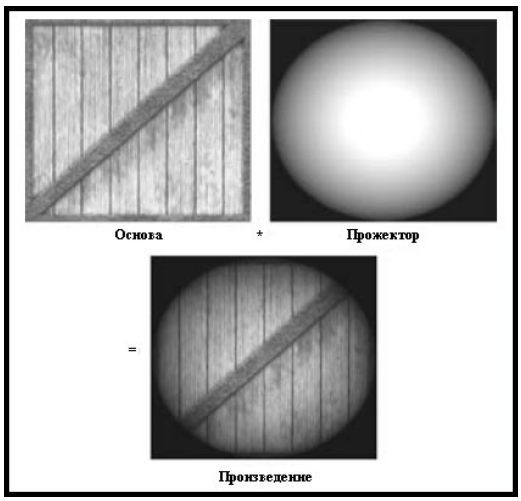

Рис. 18.1. *Визуализация освещенного прожектором ящика с использованием мультитекстурирования. Здесь мы комбинируем две текстуры, перемножая их соответствующие тексели*

ПРИМЕЧАНИЕ Как и в примере смешивания из главы 7, получаемое в результате изображение зависит от того, как именно смешиваются текстуры. В фиксированных функциях этапа мультитекстурирования формула смешивания задавалась с помощью относящихся к текстурированию режимов визуализации. В пиксельных шейдерах мы пишем функцию смешивания в коде шейдера в виде простого математического выражения. Второй подход позволяет выполнять смешивание текстур любым желаемым способом. Мы более подробно поговорми о смешивании текстур, когда будем обсуждать в этой главе пример приложения.

Смешивание текстур (в данном примере двух) для освещения ящика имеет два преимущества по сравнению с использованием модели освещения Direct3D:

- Освещение заранее вычисляется в карте освещения прожектора. Следовательно, во время выполнения освещение рассчитывать не нужно, что экономит время процессора. Конечно же, заранее можно вычислить освещение только для статических объектов и статических источников света.
- Поскольку карты освещения вычисляются заранее, мы можем использовать более точные и сложные модели освещения, чем модель Direct3D. (Чем лучше результат освещения, тем реалистичнее выглядит сцена.)

ПРИМЕЧАНИЕ Этап мультитекстурирования обычно используется для реализации всего механизма освещения статических объектов. Например, у нас может быть карта текстур, хранящая цвета объекта, такая как карта текстур ящика. Затем мы можем создать карту рассеянного освещения, хранящую оттенки рассеиваемого поверхностями света, отдельную карту отраженного света для хранения оттенков отражаемого поверхностями света, карту тумана, определяющую насколько поверхности будут скрыты туманом, и карту деталей, хранящую небольшие, часто встречающиеся на поверхностях детали. Когда все эти текстуры объединяются, мы выполняем освещение окрашивание и добавление деталей для сцены используя только просмотр предварительно созданных текстур.

ПРИМЕЧАНИЕ Карта освещения прожектора — это простейший тривиальный пример карты освещения. Обычно используются специальные программы, генерирующие карты освещения на основе сцены и источников света. Генерация карт освещения выходит за рамки этой книги. Интересующихся читателей мы отсылаем к описанию карт освещения в книге Алана Ватта и Фабио Поликарпо «3D Games: Real-time Rendering and Software Technology».

18.1.1 Разрешение работы с несколькими текстурами

Вспомните, что текстуры устанавливаются с помощью метода **IDirect3DDevice9::SetTexture**, а режимы выборки (sampler state) устанавливаются с помощью метода **IDirect3DDevice9::SetSamplerState**, чьи прототипы выглядят так:

```
HRESULT IDirect3DDevice9::SetTexture(
     DWORD Stage, // индекс этапа текстурирования
     IDirect3DBaseTexture9 *pTexture
);

HRESULT IDirect3DDevice9::SetSamplerState(
     DWORD Sampler, // индекс этапа выборки
     D3DSAMPLERSTATETYPE Type,
     DWORD Value
);
```

ПРИМЕЧАНИЕ Этап выборки с индексом *i* связвн с *i*-ым этапом текстурирования. То есть *i*-ый этап выборки задает режимы выборки для *i*-ой установленной текстуры.

Индекс этапа текстурирования/выборки идентифицирует этап текстурирования/выборки для которого мы устанавливаем текстуру или режим выборки. Следовательно, мы можем разрешить использование нескольких текстур и установить для каждой из них требуемые режимы выборки, указывая различные индексы этапов. Ранее в этой книге мы всегда указывали индекс 0, задавая первый этап, поскольку использовали только одну текстуру одновременно. Если, например, нам надо разрешить использование трех текстур, мы используем этапы 0, 1 и 2, как показано ниже:

```
// Устанавливаем первую текстуру и режимы выборки для нее
Device->SetTexture(     0, Tex1);
Device->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
Device->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
Device->SetSamplerState(0, D3DSAMP_MIPFILTER, D3DTEXF_LINEAR);

// Устанавливаем вторую текстуру и режимы выборки для нее
Device->SetTexture(     1, Tex2);
Device->SetSamplerState(1, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
Device->SetSamplerState(1, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
Device->SetSamplerState(1, D3DSAMP_MIPFILTER, D3DTEXF_LINEAR);

// Устанавливаем третью текстуру и режимы выборки для нее
Device->SetTexture(     2, Tex3);
Device->SetSamplerState(2, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
Device->SetSamplerState(2, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
Device->SetSamplerState(2, D3DSAMP_MIPFILTER, D3DTEXF_LINEAR);
```

Данный код разрешает использование трех текстур, **Tex1**, **Tex2** и **Tex3**, и устанавливает режимы фильтрации для каждой текстуры.

18.1.2 Координаты для нескольких текстур

Вспомните, в главе 6 говорилось, что для каждого трехмерного треугольника мы определяем соответствующий треугольный фрагмент текстуры, который накладывается на трехмерный треугольник. Мы делаем это путем добавления к данным каждой вершины координат текстуры. Таким образом, три вершины определяющие треугольник, определяют и соответствующий треугольный фрагмент текстуры.

Поскольку теперь мы используем несколько текстур, для каждой из трех определяющих треугольник вершин нам требуется определение соответствующего треугольного фрагмента каждой из используемых текстур. Мы делаем это добавляя наборы координат текстуры в данные каждой вершины — по одному набору координат для каждой используемой текстуры. Например, если мы хотим смешивать вместе три текстуры, то у каждой вершины должны быть три набора координат текстуры, индексирующих три используемые текстуры. Таким образом, структура данных вершины для мультитекстурирования с использованием трех текстур может выглядеть так:

```
struct MultiTexVertex
{
     MultiTexVertex(float x, float y, float z,
                    float u0, float v0,
                    float u1, float v1,
                    float u2, float v2)
     {
          _x = x; _y = y; _z = z;
          _u0 = u0; _v0 = v0;
          _u1 = u1; _v1 = v1;
          _u2 = u2; _v2 = v2;
     }

     float _x, _y, _z;
     float _u0, _v0; // Координаты текстуры накладываемой на этапе 0.
     float _u1, _v1; // Координаты текстуры накладываемой на этапе 1.
     float _u2, _v2; // Координаты текстуры накладываемой на этапе 2.

     static const DWORD FVF;
};
const DWORD MultiTexVertex::FVF = D3DFVF_XYZ | D3DFVF_TEX3;
```

Обратите внимание на использованный в настраиваемом формате вершин флаг **D3DFVF_TEX3**, который указывает, что структура вершины содержит три набора координат текстур. Фиксированный конвейер поддерживает до восьми наборов координат текстур. Если этого вам мало, придется использовать объявление вершин и программируемый конвейер.

ПРИМЕЧАНИЕ В последних версиях пиксельных шейдеров можно использовать один набор координат текстуры для индексирования сразу нескольких текстур, что позволяет убрать дополнительные наборы координат текстур из данных вершины. Конечно, в этом случае подразумевается, что на каждом этапе текстурирования используются одни и те же координаты текстуры. Если координаты текстуры на разных этапах текстурирования различаются, надо использовать несколько наборов координат текстур в данных вершины.

18.2 Входные и выходные данные пиксельного шейдера

В вершинный шейдер передаются две вещи: цвет и координаты текстуры. Обе — для пиксела.

ПРИМЕЧАНИЕ Вспомните, что цвета пикселов грани получаются путем интерполяции цветов вершин.

Координаты текстуры для пиксела это просто пара координат (u, v), определяющая тексел текстуры, который будет наложен на данный пиксел. Direct3D перед входом в пиксельный шейдер вычисляет цвет и координаты текстуры для каждого пиксела на основании цветов и координат текстуры вершин. Количество цветов и координат текстур, передаваемых в пиксельный шейдер зависит от того, сколько цветов и координат текстур возвращает вершинный шейдер. Например, если вершинный шейдер возвращает два цвета и три набора координат текстур, то Direct3D вычислит для каждого пикселя два цвета и три набора координат текстуры, которые и передаст в пиксельный шейдер. Мы отображаем переданные цвета и координаты текстур на переменные шейдера с помощью синтаксиса с двоеточием. Для рассматриваемого примера мы могли бы написать:

```
struct PS_INPUT
{
     vector c0 : COLOR0;
     vector c1 : COLOR1;
     float2 t0 : TEXCOORD0;
     float2 t1 : TEXCOORD1;
     float2 t2 : TEXCOORD2;
};
```

Возвращает пиксельный шейдер единственное вычисленное значение цвета пикселя:

```
struct PS_OUTPUT
{
     vector finalPixelColor : COLOR0;
};
```

18.3 Этапы работы с пиксельным шейдером

Приведенный ниже список перечисляет действия, которые необходимо выполнить для создания и использования пиксельного шейдера.

1. Написать и скомпилировать пиксельный шейдер.
2. Создать представляющий пиксельный шейдер интерфейс **IDirect3DPixelShader9** на основании скомпилированного кода шейдера.
3. Установить пиксельный шейдер с помощью метода **IDirect3DDevice9::SetPixelShader**.

Конечно же, после завершения работы нам надо уничтожить пиксельный шейдер. В следующих разделах детально рассмотрены все эти этапы.

18.3.1 Написание и компиляция пиксельного шейдера

Мы компилируем пиксельный шейдер точно так же, как компилировали вершинные шейдеры. Сперва мы должны написать программу пиксельного шейдера. В этой книге мы пишем наши шейдеры на HLSL. Как только исходный код шейдера написан, мы можем скомпилировать его с помощью функции **D3DXCompileShaderFromFile**, как описано в разделе 16.2. Вспомните, что эта функция возвращает указатель на интерфейс **ID3DXBuffer**, который содержит скомпилированный код шейдера.

ПРИМЕЧАНИЕ Поскольку сейчас мы работаем с пиксельными шейдерами, важно помнить о том, что при компиляции необходимо указывать целевую версию пиксельных шейдероы (т.е. **ps_2_0**), а не целевую версию вершинных шейдеров (т.е. **vs_2_0**). Целевая версия шейдеров указывается в параметре функции **D3DXCompileShaderFromFile**. За подробностями обратитесь к разделу 16.2.

18.3.2 Создание пиксельного шейдера

Когда у нас есть скомпилированный код шейдера, мы можем получить указатель на представляющий пиксельный шейдер интерфейс **IDirect3DPixelShader9**, воспользовавшись следующим методом:

```
HRESULT IDirect3DDevice9::CreatePixelShader(
     CONST DWORD *pFunction,
     IDirect3DPixelShader9** ppShader
);
```

- **pFunction** — Указатель на скомпилированный код шейдера.
- **ppShader** — Возвращает указатель на интерфейс **IDirect3DPixelShader9**.

Предположим, например, что переменная **shader** — это экземпляр интерфейса **ID3DXBuffer**, который содержит скомпилированный код шейдера. Тогда, для получения интерфейса **IDirect3DPixelShader9** следует написать:

```
IDirect3DPixelShader9* MultiTexPS = 0;
hr = Device->CreatePixelShader(
           (DWORD*)shader->GetBufferPointer(),
           &MultiTexPS);
```

ПРИМЕЧАНИЕ Повторим еще раз, **D3DXCompileShaderFromFile** — это функция, которая возвращает скомпилированный код шейдера (**shader**).

18.3.3 Установка пиксельного шейдера

После того, как мы получили указатель на представляющий наш пиксельный шейдер интерфейс **IDirect3DPixelShader9**, можно разрешить его использование с помощью следующего метода:

```
HRESULT IDirect3DDevice9::SetPixelShader(
     IDirect3DPixelShader9* pShader
);
```

Метод получает единственный параметр в котором мы передаем указатель на устанавливаемый пиксельный шейдер. Чтобы включить шейдер, который мы создали в разделе 18.3.2, следует написать:

```
Device->SetPixelShader(MultiTexPS);
```

18.3.4 Уничтожение пиксельного шейдера

Как и для всех интерфейсов Direct3D, здесь для очистки мы должны при завершении работы с интерфейсом вызвать его метод **Release**. Продолжая изучать пример пиксельного шейдера созданного в разделе 18.3.2, мы получаем:

```
d3d::Release<IDirect3DPixelShader9*>(MultiTexPS);
```

18.4 Объекты выборки в HLSL

Выборка для текстур в пиксельном шейдере осуществляется с помощью специальных встроенных функций HLSL **tex***.

ПРИМЕЧАНИЕ Под *выборкой* (*sampling*) мы понимаем выборку по индексу соответствующего пикселю текселя на основе координат текстуры для пикселя и режимов выборки (режимов фильтрации текстур).

В общем случае эти функции требуют, чтобы мы указали две вещи:

- Координаты текстуры (u, v), используемые для индексации элементов текстуры.
- Текстуру в которой выполняется индексация.

Координаты текстуры (u, v), как вы помните, передаются во входных данных пиксельного шейдера. Текстура, элемент которой нам надо получить, идентифицируется в пиксельном шейдере с помощью специального объекта HLSL, называемого *объект выборки* (*sampler*). Можно думать об объектах выборки как об объектах, идентифицирующих текстуру и этап выборки. Для примера предположим, что мы используем три этапа текстурирования, а значит в пиксельном шейдере нам требуется возможность ссылаться на каждый из этих этапов. Для этого в пиксельном шейдере напишем:

```
sampler FirstTex;
sampler SecondTex;
sampler ThirdTex;
```

Direct3D свяжет каждый из этих объектов выборки с уникальным этапом выборки. Затем в приложении мы определяем этап, которому соответствует объект выборки и устанавливаем для этого этапа требуемые текстуру и режимы выборки. Приведенный ниже код показывает, как приложение может установить текстуру и режимы выборки для **`FirstTex`**:

```
// Создание текстуры
IDirect3DTexture9* Tex;
D3DXCreateTextureFromFile(Device, "tex.bmp", &Tex);
.
.
.
// Получение дескриптора константы
FirstTexHandle = MultiTexCT->GetConstantByName(0, "FirstTex");

//Получение описания константы
D3DXCONSTANT_DESC FirstTexDesc;
UINT count;
MultiTexCT->GetConstantDesc(FirstTexHandle, &FirstTexDesc, &count);
.
.
.
// Устанавливаем текстуру/режимы выборки для объекта выборки
// FirstTex. Мы определяем этап с которым связан FirstTex по значению
// члена данных D3DXCONSTANT_DESC::RegisterIndex:
Device->SetTexture(     FirstTexDesc.RegisterIndex, Tex);
Device->SetSamplerState(FirstTexDesc.RegisterIndex,
                        D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
Device->SetSamplerState(FirstTexDesc.RegisterIndex,
                        D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
Device->SetSamplerState(FirstTexDesc.RegisterIndex,
                        D3DSAMP_MIPFILTER, D3DTEXF_LINEAR);
```

ПРИМЕЧАНИЕ Альтернативным способом, который может применяться вместо использования объектов выборки, является применение более специализированных и строго типизированных типов **`sampler1D`**, **`sampler2D`**, **`sampler3D`** и **`samplerCube`**. Эти типы обеспечивают большую безопасность типов и гарантируют, что будут использоваться только соответствующие функции **`tex*`**. Например, для объекта **`sampler2D`** могут применяться только функции **`tex2D*`**. Аналогично, для объекта **`sampler3D`** могут применяться только функции **`tex3D*`**.

18.5 Пример приложения: мультитекстурирование в пиксельном шейдере

Пример приложения для данной главы демонстрирует мультитекстурирование с использованием пиксельных шейдеров. Пример формирует текстурированный квадрат, отмеченный на рис. 18.2 как «результат» путем смешивания текстуры ящика, текстуры прожектора и текстуры со строкой «Pixel Shader Sample».

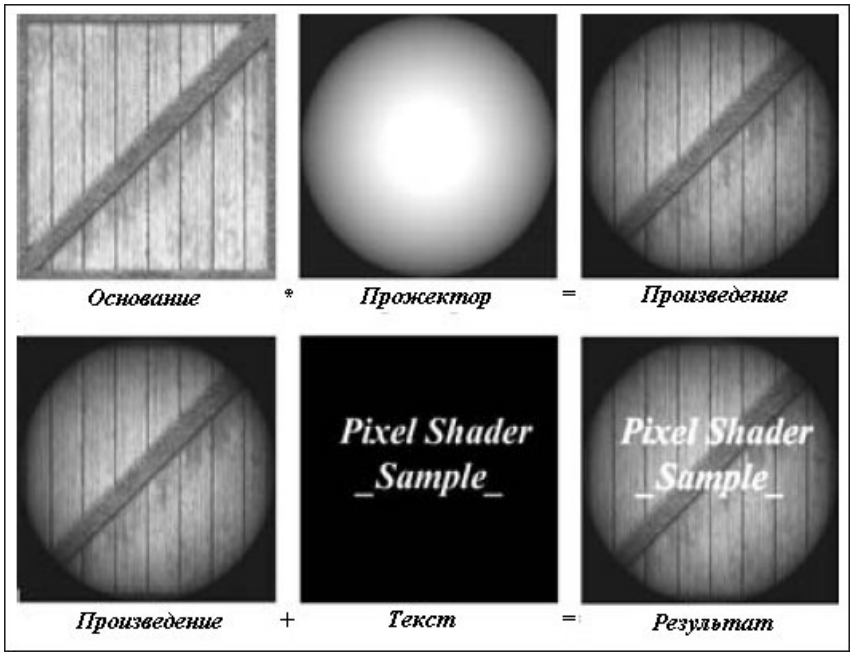

Рис. 18.2. *Комбинирование текстур. Пусть* ***b****,* ***s*** *и* ***t*** *— это цвета соответствующих текселей из текстуры ящика, текстуры прожектора и текстуры текста соответственно. Тогда цвет их комбинации определяется по формуле* $\boldsymbol{c} = \boldsymbol{b} \otimes \boldsymbol{s} + \boldsymbol{t}$*, где* $\otimes$ *обозначает покомпонентное умножение*

Данный пример может быть реализован и без пиксельных шейдеров. Однако, это более простой и прямолинейный способ реализации, позволяющий к тому же продемонстрировать написание, создание и использование пиксельных шейдеров без отвлечения на реализацию алгоритма какого-нибудь специального эффекта.

Хотя в рассматриваемом примере мы используем одновременно только три текстуры, весьма полезно узнать сколько объектов выборки могут одновремено использоваться в каждой из версий пиксельных шейдеров. Другими словами, как количество одновременно используемых текстур зависит от используемой версии пиксельных шейдеров.

- Пиксельные шейдеры версий от ps_1_1 до ps_1_3 поддерживают до четырех выборок текстуры.
- Пиксельные шейдеры версии ps_1_4 поддерживают до шести выборок тектсуры.
- Пиксельные шейдеры версий от ps_2_0 до ps_3_0 поддерживают до 16 выборок текстуры.

Код пиксельного шейдера для реализации мультитекстурирования с использованием трех текстур выглядит следующим образом:

```
//
// Файл    : ps_multitex.txt
// Описание: Пиксельный шейдер, выполняющий мультитекстурирование
//

//
// Глобальные переменные
//

sampler BaseTex;
sampler SpotLightTex;
sampler StringTex;

//
// Структуры
//

struct PS_INPUT
{
     float2 base      : TEXCOORD0;
     float2 spotlight : TEXCOORD1;
     float2 text      : TEXCOORD2;
};

struct PS_OUTPUT
{
     vector diffuse : COLOR0;
};

//
// Точка входа
//

PS_OUTPUT Main(PS_INPUT input)
{
     // Обнуляем члены выходной структуры
     PS_OUTPUT output = (PS_OUTPUT)0;

     // Выборка данных из соответствующих текстур
     vector b = tex2D(BaseTex, input.base);
     vector s = tex2D(SpotLightTex, input.spotlight);
     vector t = tex2D(StringTex, input.text);
```

```
    // Комбинирование цветов текселей
    vector c = b * s + t;

    // Слегка увеличиваем яркость пикселя
    c += 0.1f;

    // Сохраняем результатирующий цвет
    output.diffuse = c;

    return output;
}
```

Сперва в пиксельном шейдере мы объявляем три объекта выборки — по одному для каждой, участвующей в смешивании текстуры. Затем мы описываем входную и выходную структуры. Обратите внимание, что в пиксельный шейдер не передается никаких значений цветов; это вызвано тем, что для текстурирования и освещения объекта применяются только текстуры. Так, **`BaseTex`** хранит цвета нашей поверхности, а **`SpotLightTex`** — карту освещения. Пиксельный шейдер возвращает единственное значение цвета, которое определяет вычисленный нами цвет данного пикселя.

Функция **`Main`** выполняет выборку значений для трех текстур с помощью функции **`tex2D`**. Таким образом, мы получаем отображаемые на обрабатываемый в данный момент пиксель тексели каждой из текстур, определяемые на основе заданных координат текстуры и объекта выборки. Затем мы комбинируем цвета текселей согласно формуле **`c = b * s + t`**. После этого мы слегка осветляем полученный пиксель, добавляя 0.1f к каждой из его компонент. И, наконец, мы сохраняем цвет полученного в результате пикселя и взвращаем его.

Теперь, посмотрев на код пиксельного шейдера, мы готовы переключить передачу и отправиться к коду приложения. К рассматриваемой нами теме относятся следующие глобальные переменные приложения:

```
IDirect3DPixelShader9* MultiTexPS = 0;
ID3DXConstantTable* MultiTexCT    = 0;

IDirect3DVertexBuffer9* QuadVB    = 0;

IDirect3DTexture9* BaseTex        = 0;
IDirect3DTexture9* SpotLightTex   = 0;
IDirect3DTexture9* StringTex      = 0;

D3DXHANDLE BaseTexHandle          = 0;
D3DXHANDLE SpotLightTexHandle     = 0;
D3DXHANDLE StringTexHandle        = 0;

D3DXCONSTANT_DESC BaseTexDesc;
D3DXCONSTANT_DESC SpotLightTexDesc;
D3DXCONSTANT_DESC StringTexDesc;
```

Структура данных вершины для примера мультитекстурирования выглядит следующим образом:

```
struct MultiTexVertex
{
     MultiTexVertex(float x, float y, float z,
                    float u0, float v0,
                    float u1, float v1,
                    float u2, float v2)
     {
          _x = x; _y = y; _z = z;
          _u0 = u0; _v0 = v0;
          _u1 = u1; _v1 = v1;
          _u2 = u2, _v2 = v2;
     }

     float _x, _y, _z;
     float _u0, _v0;
     float _u1, _v1;
     float _u2, _v2;

     static const DWORD FVF;
};
const DWORD MultiTexVertex::FVF = D3DFVF_XYZ | D3DFVF_TEX3;
```

Обратите внимание, что она содержит три набора координат текстур.

Функция **`Setup`** выполняет следующие действия:

- Заполняет вершинный буфер данными квадрата.
- Компилирует пиксельный шейдер.
- Создает пиксельный шейдер.
- Загружает текстуры.
- Устанавливает матрицу проекции и отключает освещение.
- Получает дескрипторы объектов выборки.
- Получает описания объектов выборки.

```
bool Setup()
{
     HRESULT hr = 0;

     //
     // Создание квадрата
     //

     Device->CreateVertexBuffer(
          6 * sizeof(MultiTexVertex),
          D3DUSAGE_WRITEONLY,
          MultiTexVertex::FVF,
          D3DPOOL_MANAGED,
          &QuadVB,
          0);

     MultiTexVertex* v = 0;
     QuadVB->Lock(0, 0, (void**)&v, 0);
```

```
    v[0] = MultiTexVertex(-10.0f, -10.0f, 5.0f,
                          0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0f);
    v[1] = MultiTexVertex(-10.0f, 10.0f, 5.0f,
                          0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f);
    v[2] = MultiTexVertex( 10.0f, 10.0f, 5.0f,
                          1.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f);

    v[3] = MultiTexVertex(-10.0f, -10.0f, 5.0f,
                          0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0f);
    v[4] = MultiTexVertex( 10.0f, 10.0f, 5.0f,
                          1.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f);
    v[5] = MultiTexVertex( 10.0f, -10.0f, 5.0f,
                          1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f);

    QuadVB->Unlock();

//
// Компиляция шейдера
//
    ID3DXBuffer* shader = 0;
    ID3DXBuffer* errorBuffer = 0;

    hr = D3DXCompileShaderFromFile(
         "ps_multitex.txt",
         0,
         0,
         "Main", // имя точки входа
         "ps_1_1",
         D3DXSHADER_DEBUG,
         &shader,
         &errorBuffer,
         &MultiTexCT);

    // Выводим любые сообщения об ошибках
    if(errorBuffer)
    {
         ::MessageBox(0, (char*)errorBuffer->GetBufferPointer(),
                      0, 0);
         d3d::Release<ID3DXBuffer*>(errorBuffer);
    }

    if(FAILED(hr))
    {
         ::MessageBox(0, "D3DXCompileShaderFromFile() - FAILED",
                      0, 0);
         return false;
    }

    //
    // Создание пиксельного шейдера
    //

    hr = Device->CreatePixelShader(
         (DWORD*)shader->GetBufferPointer(),
         &MultiTexPS);
```

```
if(FAILED(hr))
{
     ::MessageBox(0, "CreateVertexShader - FAILED", 0, 0);
     return false;
}

d3d::Release<ID3DXBuffer*>(shader);

//
// Загрузка текстур
//

D3DXCreateTextureFromFile(Device, "crate.bmp", &BaseTex);
D3DXCreateTextureFromFile(Device, "spotlight.bmp",
                                          &SpotLightTex);
D3DXCreateTextureFromFile(Device, "text.bmp", &StringTex);

//
// Установка матрицы проекции
//

D3DXMATRIX P;
D3DXMatrixPerspectiveFovLH(
     &P, D3DX_PI * 0.25f,
     (float)Width / (float)Height, 1.0f, 1000.0f);

Device->SetTransform(D3DTS_PROJECTION, &P);

//
// Запрещение освещения
//

Device->SetRenderState(D3DRS_LIGHTING, false);

//
// Получение дескрипторов
//

BaseTexHandle = MultiTexCT->GetConstantByName(0, "BaseTex");
SpotLightTexHandle = MultiTexCT->GetConstantByName(0,
                                            "SpotLightTex");
StringTexHandle = MultiTexCT->GetConstantByName(0, "StringTex");

//
// Получение описания констант
//

UINT count;
MultiTexCT->GetConstantDesc(
     BaseTexHandle,
     &BaseTexDesc,
     &count);
MultiTexCT->GetConstantDesc(
     SpotLightTexHandle,
     &SpotLightTexDesc,
     &count);
```

```
     MultiTexCT->GetConstantDesc(
          StringTexHandle,
          &StringTexDesc,
          &count);

     MultiTexCT->SetDefaults(Device);

     return true;
}
```

Функция **`Display`** устанавливает пиксельный шейдер, разрешает использование трех текстур и устанавливает для них требуемые режимы выборки перед визуализацией квадрата.

```
bool Display(float timeDelta)
{
     if(Device)
     {
          //
          // ...код обновления камеры пропущен
          //

          //
          // Визуализация
          //

          Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                        0xffffffff, 1.0f, 0);
          Device->BeginScene();

          // Установка пиксельного шейдера
          Device->SetPixelShader(MultiTexPS);
          Device->SetFVF(MultiTexVertex::FVF);
          Device->SetStreamSource(0, QuadVB, 0,
                                  sizeof(MultiTexVertex));

          // Базовая текстура
          Device->SetTexture(BaseTexDesc.RegisterIndex, BaseTex);
          Device->SetSamplerState(BaseTexDesc.RegisterIndex,
                                  D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
          Device->SetSamplerState(BaseTexDesc.RegisterIndex,
                                  D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
          Device->SetSamplerState(BaseTexDesc.RegisterIndex,
                                  D3DSAMP_MIPFILTER, D3DTEXF_LINEAR);

          // Текстура прожектора
          Device->SetTexture(SpotLightTexDesc.RegisterIndex,
                                                    SpotLightTex);
          Device->SetSamplerState(SpotLightTexDesc.RegisterIndex,
                                  D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
          Device->SetSamplerState(SpotLightTexDesc.RegisterIndex,
                                  D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
          Device->SetSamplerState(SpotLightTexDesc.RegisterIndex,
                                  D3DSAMP_MIPFILTER, D3DTEXF_LINEAR);
```

```
            // Текстура с текстом
            Device->SetTexture( StringTexDesc.RegisterIndex,
                                                        StringTex);
            Device->SetSamplerState(StringTexDesc.RegisterIndex,
                                    D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
            Device->SetSamplerState(StringTexDesc.RegisterIndex,
                                    D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
            Device->SetSamplerState(StringTexDesc.RegisterIndex,
                                    D3DSAMP_MIPFILTER, D3DTEXF_LINEAR);

            // Рисуем квадрат
            Device->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 2);

            Device->EndScene();
            Device->Present(0, 0, 0, 0);
        }
        return true;
}
```

И, конечно, следует помнить о необходимости освобождения полученных интерфейсов в функции **`Cleanup`**:

```
void Cleanup()
{
     d3d::Release<IDirect3DVertexBuffer9*>(QuadVB);

     d3d::Release<IDirect3DTexture9*>(BaseTex);
     d3d::Release<IDirect3DTexture9*>(SpotLightTex);
     d3d::Release<IDirect3DTexture9*>(StringTex);

     d3d::Release<IDirect3DPixelShader9*>(MultiTexPS);
     d3d::Release<ID3DXConstantTable*>(MultiTexCT);
}
```

18.6 Итоги

- Пиксельные шейдеры заменяют этап мультитекстурирования в фиксированном конвейере функций. Более того, пиксельные шейдеры предоставляют возможность изменять цвет каждого отдельного пикселя любым способом и предоставляют доступ к данным текстуры, что позворяет реализовать множество специальных эффектов, которые недоступны при работе с фиксированным конвейером функций.
- Мультитекстурирование — это процесс одновременного наложения нескольких текстур и их смешивания для получения желаемого результата. Мультитекстурирование обычно используется для реализации всего механизма освещения статических объектов.
- Встроенный в HLSL объект **`sampler`** идентифицирует конкретный этап текстурирования/выборки. Объект **`sampler`** используется для ссылки на этап текстурирования/выборки из пиксельного шейдера.

ПРИМЕЧАНИЕ Как только вы разберетесь с принципами работы вершинных и пиксельных шейдеров, у вас появится множество идей об эффектах, которые можно реализовать с их помощью. Наилучший способ генерации идей, которые могут быть реализованы с помощью вершинных и пиксельных шейдеров — изучение уже реализованных эффектов. Изданная Wordware Publishing книга «Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks», редактором которой является Вольфганг Энджел, является хорошей отправной точкой. Кроме того следует посетить сайты для разработчиков Nvidia и ATI, расположенные по адресам *http://developer.nvidia.com/* и *http://ati.com/developer/index.html*, соответственно. Также рекомендуем вам книгу Рэндайма Фернандо и Марка Дж. Килгарда «CG: The Cg Tutorial». Она является замечательным учебником по программированию трехмерной графики с использованием высокоуровневого языка графического программирования Cg, который во многом похож на Direct3D HLSL.

Глава 19
Каркас эффектов

Графический эффект обычно состоит из нескольких компонентов: вершинный и/или пиксельный шейдер, список режимов устройства, которые должны быть установлены, и один или несколько проходов визуализации. Кроме того, часто для графических эффектов желательно наличие механизма обработки сбоев, позволяющего эффекту выполняться на различных видеокартах Логичным шагом будет попытка объединения всех этих задач в один блок.

Каркас эффектов Direct3D предоставляет подобный механизи для объединения связанных с визуальным эффектом задач в один файл эффекта. Такая реализация эффектов имеет ряд преимуществ. Во-первых, она позволяет нам менять реализацию эффекта без перекомпиляции исходного кода приложения. Во-вторых, мы объединяем все компоненты эффекта в одном файле.

Данная глава предоставит вам необходимую информацию для написания и использования файла эффекта. Обратите внимание, что файл эффекта представляет собой обычный текстовый файл ASCII, точно такой же как и программы на HLSL.

Цели

- Получить представление о структуре и организации файла эффекта.
- Узнать о некоторых дополнительных встроенных объектах HLSL.
- Узнать как в файле эффектов указываются режимы устройства.
- Научиться создавать и использовать эффекты.
- Приобрести опыт работы с каркасом эффектов, изучив примеры программ.

19.1 Техники и проходы

Файл эффекта состоит из одной или нескольких *техник* (*techniques*). Техникой называется конкретный способ реализации спецэффекта. Другими словами, файл эффекта описывает один или несколько способов реализации одного и того же спецэффекта. Зачем надо несколько различных реализаций одного и того же эффекта? Дело в том, что установленное на компьютере оборудование может не поддерживать определенную реализацию эффекта. Следовательно необходимо реализовать несколько версий одного и того же эффекта, ориентированных на различное оборудование.

ПРИМЕЧАНИЕ Например, мы можем реализовать две версии одного и того же эффекта — одну с использованием шейдеров, а другую с использованием фиксированного конвейера. В этом случае те пользователи, чьи видеокарты поддерживают шейдеры, получат преимущества реализации, использующей шейдеры, а те пользователи, чьи видеокарты шейдеры не поддерживают могут продолжать использовать фиксированный конвейер.

Возможность реализовать все версии эффекта в одном файле дает нам более полную инкапсуляцию эффекта в целом, а объединение реализации эффекта это и есть одна из целей каркаса эффектов.

Каждая техника объединяет один или несколько *проходов визуализации* (*rendering passes*). Проход визуализации объединяет режимы устройства, режимы выборки и шейдеры, используемые для визуализации на данном этапе.

ПРИМЕЧАНИЕ Файл эффектов не ограничивает вас только использованием программируемого конвейера. Например, вы можете использовать фиксированный конвейер для управления режимами устройств, такими как освещение, материалы и текстуры.

Причина наличия нескольких проходов в том, что для реализации некоторых эффектов необходимо визуализировать один и тот же объект несколько раз с различными режимами визуализации, шейдерами и т.д. для каждого прохода. Вспомните, например, как в главе 8 чтобы получить эффект отражения мы несколько раз визуализировали один и тот же объект с различными состояниями устройства.

В качестве примера приведем скелет файла эффектов с двумя техниками, где первая техника состоит из одного прохода, а вторая — из двух:

```
// effect.txt
...
technique T0
{
     // Первый и единственный проход для данной техники
     pass P0
     {
     ...[указываем состояния устройства, шейдеры, выборки и т.д.
         для прохода]
```

```
    }
}

technique T1
{
    // Первый проход
    pass P0
    {
    ...[указываем состояния устройства, шейдеры, выборки и т.д.
        для прохода]
    }

    // Второй проход
    pass P1
    {
    ...[указываем состояния устройства, шейдеры, выборки и т.д.
        для прохода]
    }
}
```

19.2 Встроенные объекты HLSL

Сейчас мы обсудим несколько встроенных объектных типов HLSL. Мы не обсуждали их раньше потому что они используются главным образом в каркасе эффектов.

19.2.1 Объекты текстуры

Встроенный тип HLSL **`texture`** представляет объект **`IDirect3DTexture9`**. Используя объект **`texture`** мы можем связывать текстуру с заданным этапом выборки непосредственно в файле эффекта. У объекта **`texture`** есть следующие доступные члены данных:

- **`type`** — Тип текстуры (т.е., 2D, 3D).
- **`format`** — Формат пикселей текстуры.
- **`width`** — Ширина текстуры в пикселях.
- **`height`** — Высота текстуры в пикселях.
- **`depth`** — Глубина (для трехмерных объемных текстур) текстуры в пикселях.

ПРИМЕЧАНИЕ До сих пор мы использовали текстуры только для хранения изображений, но познакомившись с провессиональными техниками вы обнаружите, что текстуры применяются для хранения произвольной табличной информации. Другими словами, текстура — это просто таблица с данными; совсем необязательно, чтобы она хранила изображение. Например, в *рельефном текстурировании* (*bump mapping*) используются *карты нормалей* (*normal map*), представляющие собой текстуры, содержащие векторы нормалей для каждого элемента.

19.2.2 Объекты выборки и режимы выборки

Мы обсуждали объекты выборки в главе 18; однако каркас эффектов добавляет новое ключевое слово **sampler_state**. С помощью ключевого слова **sampler_state** мы можем инициализировать объект **sampler** (то есть, устанавливать текстуру и режимы выборки для объекта выборки непосредственно из файла эффкетов). Следующий фрагмент кода иллюстрирует эту возможность:

```
Texture Tex;

sampler S0 = sampler_state
{
     Texture   = (Tex);
     MinFilter = LINEAR;
     MagFilter = LINEAR;
     MipFilter = LINEAR;
};
```

Здесь мы связываем текстуру **Tex** с этапом текстурирования, которому соответствует объект **S0**, а также задаем режимы выборки для этапа выборки, которому соответствует **S0**. Все эти параметры мы явно устанавливаем непосредственно из файла эффектов!

19.2.3 Объекты вершинных и пиксельных шейдеров

Встроенные типы HLSL **vertexshader** и **pixelshader** представляют вершинные и пиксельные шейдеры соответственно. Они используются в каркасе эффектов для ссылки на конкретный вершинный и/или пиксельный шейдер, который должен использоваться в данном проходе визуализации. Типы **vertexshader** и **pixelshader** могут инициализироваться из приложения через интерфейс **ID3DXEffect** с помощью методов **ID3DX-Effect::SetVertexShader** и **ID3DXEffect::SetPixelShader** соответственно. Например, пусть **Effect** — это корректный объект **ID3DXEffect**, **VS** — это корректный объект **IDirect3DVertexShader9** и **VSHandle** — это значение типа **D3DXHANDLE**, которое ссылается на объект **vertexshader** в файле эффекта; тогда мы можем инициализировать вершинный шейдер, на который ссылается **VSHandle** написав:

```
Effect->SetVertexShader(VSHandle, VS);
```

Мы больше узнаем о методах **SetVertexShader** и **SetPixelShader** когда будем обсуждать инициализацию переменных файла эффекта из приложения.

Кроме того, мы можем написать вершинный и/или пиксельный шейдер непосредственно в файле эффекта. Затем, используя специальный синтаксис компиляции мы можем инициализировать переменную шейдера. Приведенный

ниже фрагмент кода показывает инициализацию переменной **ps** типа **pixelshader**.

```
// Определение Main:
OUTPUT Main(INPUT input){...}

// Компиляция Main:
pixelshader ps = compile ps_2_0 Main();
```

Обратите внимание, что после ключевого слова **compile** мы указываем версию шейдеров, а за ней — имя точки входа шейдера. Заметьте, что при использовании такого стиля для инициализации объектов вершинных/пиксельных шейдеров, функция, являющаяся точкой входа, должна быть определена в файле эффекта.

И, наконец, мы связываем шейдер с конкретным проходом, как показано ниже:

```
// Определение Main:
OUTPUT Main(INPUT input){...}

// Компиляция Main:
vertexshader vs = compile vs_2_0 Main();

pass P0
{
     // Устанавливаем vs в качестве
     // вершинного шейдера для данного прохода
     vertexshader = (vs);

     ...
}
```

Или в более компактной форме:

```
pass P0
{
     // Устанавливаем вершинный шейдер с точкой входа Main()
     // в качестве вершинного шейдера для данного прохода
     vertexShader = compile vs_2_0 Main();

     ...
}
```

ПРИМЕЧАНИЕ

Следует упомянуть, чтобы вы имели представление об этой возможности, что типы **vertexshader** и **pixelshader** можно инициализировать с использованием следующего синтаксиса:

```
vertexshader vs = asm {
     /* здесь размещаются ассемблерные инструкции */
};

pixelshader  ps = asm {
     /* здесь размещаются ассемблерные инструкции */
};
```

Этот синтаксис используется если вы пишете свои шейдеры на языке ассемблера.

19.2.4 Строки

Существуют и строковые объекты, которые можно использовать следующим образом:

```
string filename = "texName.bmp";
```

Хотя строковые типы не используются в функциях HLSL, они могут быть прочитаны из приложения. Таким образом мы можем поместить в файл эффектов ссылки на файлы данных, которые данный эффект использует, например, имена файлов текстур и X-файлов.

19.2.5 Аннотации

Помимо указания способа использования о котором мы уже говорили, к переменным могут присоединяться аннотации. Аннотации никак не используются в HLSL, но к ним можно получить доступ из приложения через каркас эффектов. Аннотации используются просто для присоединения к переменным «примечаний», которые приложение хотело бы видеть связанными с данной переменной. Добавляются аннотации с использованием синтаксиса ***<аннотация>***, который иллюстрирует следующий фрагмент кода:

```
texture tex0 < string name = "tiger.bmp"; >;
```

В данном примере аннотация — это **<string name = "tiger.bmp";>**. Она связывает с переменной **tex0** строку, а именно имя файла, содержащего данные текстуры. Ясно, что добавление к текстуре аннотации с именем соответсвующего файла может оказаться полезным.

Получить аннотации можно с помощью следующего метода:

```
D3DXHANDLE ID3DXEffect::GetAnnotationByName(
     D3DXHANDLE hObject,
     LPCSTR pName
);
```

Здесь **pName** — это имя аннотации для которой мы хотим получить дескриптор, а **hObject** — это дескриптор родительского блока в котором расположена аннотация, такого как техника, проход или структурный блок. После того, как мы получили дескриптор аннотации, можно получить информацию о ней с помощью метода **ID3DXEffect::GetParameterDesc**, который заполняет структуру **D3DXCONSTANT_DESC**. За подробностями обращайтесь к документации DirectX SDK.

19.3 Состояния устройства в файле эффекта

Обычно для правильного выполнения эффекта мы должны установить состояния устройства, такие как режимы визуализации, режимы текстурирования,

материалы, освещение и текстуры. Чтобы весь эффект можно было включить в один файл, каркас эффектов позволяет нам устанавливать состояния устройства из файла эффекта. Состояния устройства устанавливаются внутри блока прохода визуализации и для этого используется следующий синтаксис:

```
Состояние = Значение;
```

Чтобы просмотреть полный список доступных состояний, поищите слово «states» в алфавитном указателе документации DirectX SDK или на вкладке **Contents** справочной системы SDK выберите пункт DirectX Graphics\Reference\Effect Reference\Effect Format\States.

Возьмем, к примеру, состояние **FillMode**. Если вы поищете его упоминание в документации SDK, то увидите, что доступные для него значения — это те же значения, которые доступны для режима **D3DFILLMODE**, только без префикса **D3DFILL_**. Если посмотреть описание **D3DFILLMODE** в документации SDK, мы обнаружим допустимые значения **D3DFILL_POINT**, **D3DFILL_WIREFRAME**, и **D3DFILL_SOLID**. Таким образом, для файла эффектов мы отбрасываем префикс и получаем следующие допустимые значения состояния **FillMode**: **POINT**, **WIREFRAME** и **SOLID**. Например, мы можем написать в файле эффектов:

```
FillMode = WIREFRAME;
FillMode = POINT;
FillMode = SOLID;
```

ПРИМЕЧАНИЕ В последующих разделах мы будем устанавливать некоторые состояния устройства в примерах программ. В большинстве случаев назначение состояния ясно видно из его названия, но если вам потребуются дополнительные разъяснения, обратитесь к документации SDK.

19.4 Создание эффекта

Эффект представляется интерфейсом **ID3DXEffect**, который мы создаем с помощью следующей функции из библиотеки D3DX:

```
HRESULT D3DXCreateEffectFromFile(
     LPDIRECT3DDEVICE9 pDevice,
     LPCSTR pSrcFile,
     CONST D3DXMACRO* pDefines,
     LPD3DXINCLUDE pInclude,
     DWORD Flags,
     LPD3DXEFFECTPOOL pPool,
     LPD3DXEFFECT* ppEffect,
     LPD3DXBUFFER *ppCompilationErrors
);
```

- **pDevice** — Устройство, связанное с создаваемым эффектом **ID3DXEffect**.

- **pSrcFile** — Имя текстового файла (файла эффекта) содержащего исходный код того эффекта, который мы хотим скомпилировать.
- **pDefines** — Необязательный параметр, в этой книге мы всегда будем указывать в нем **null**.
- **pInclude** — Указатель на интерфейс **ID3DXInclude**. Этот интерфейс разработан для тех приложений, которым требуется переопределить устанавливаемое по умолчанию поведение включения. В общем случае поведение по умолчанию замечательно работает и поэтому мы игнорируем данный параметр, передавая в нем **null**.
- **Flags** — Необязательные флаги компиляции шейдеров из файла эффекта; если флаги не нужны, укажите 0. Можно использовать следующие значения:
  - **D3DXSHADER_DEBUG** — Приказывает компилятору включать в скомпилированный файл отладочную информацию.
  - **D3DXSHADER_SKIPVALIDATION** — Приказывает компилятору не выполнять проверку корректности кода. Этот флаг следует использовать только при работе с теми шейдерами в правильности кода которых вы абсолютно уверены.
  - **D3DXSHADER_SKIPOPTIMIZATION** — Приказывает компилятору не выполнять оптимизацию кода. Обычно этот флаг используется при отладке, когда вы не хотите, чтобы компилятор вносил какие-либо изменения в код.
- **pPool** — Необязательный указатель на интерфейс **ID3DXEffectPool**, используемый для определения того, как параметры эффекта будут совместно использоваться несколькими экземплярами эффекта. В этой книге в данном параметре мы всегда указываем **null**, сообщая тем самым что параметры не будут совместно использоваться несколькими файлами эффектов.
- **ppEffect** — Возвращает указатель на интерфейс **ID3DXEffect**, представляющий созданный эффект.
- **ppCompilationErrors** — Возвращает указатель на интерфейс **ID3DXBuffer**, содержащий строку с кодами обнаруженых при компиляции ошибок и их описанием.

Вот пример вызова функции **D3DXCreateEffectFromFile**:

```
//
// Создание эффекта
//

ID3DXEffect* Effect = 0;
ID3DXBuffer* errorBuffer = 0;
hr = D3DXCreateEffectFromFile(
      Device,           // связанное устройство
      "effect.txt",     // имя исходного файла
```

```
      0,                // нет объявлений препроцессора
      0,                // нет интерфейса ID3DXInclude
      D3DXSHADER_DEBUG, // флаги компиляции
      0,                // параметры не используются совместно
      &Effect,          // возвращает результат
      &errorBuffer);    // возвращает строку с ошибками

// Выводим любые сообщения об ошибках
if( errorBuffer )
{
     ::MessageBox(0, (char*)errorBuffer->GetBufferPointer(), 0, 0);
     d3d::Release<ID3DXBuffer*>(errorBuffer);
}

if(FAILED(hr))
{
     ::MessageBox(0, "D3DXCreateEffectFromFile() - FAILED", 0, 0);
     return false;
}
```

19.5 Установка констант

Как и в случае с вершинными и пиксельными шейдерами нам необходима возможность инициализировать переменные эффекта из приложения. Однако, вместо таблицы констант, которой мы пользовались при работе с вершинными и пиксельными шейдерами, интерфейс **ID3DXEffect** предоставляет встроенные методы для инициализации переменных. Мы не будем приводить список всех методов для инициализации различных типов переменных, поскольку он очень велик и в нем много повторений. Если вам все же хочется увидеть полный список — обратитесь к документации DirectX SDK. Вот сокращенный список методов:

Функция	Описание
HRESULT ID3DXEffect::SetFloat(D3DXHANDLE hParameter, FLOAT f);	Присваивает идентифицируемой дескриптором **hParameter** переменной с плавающей точкой из файла эффекта значение **f**
HRESULT ID3DXEffect::SetMatrix(D3DXHANDLE hParameter, CONST D3DXMATRIX* pMatrix);	Инициализирует идентифицируемую дескриптором **hParameter** матрицу в файле эффекта, копируя в нее значения из матрицы на которую указывает **pMatrix**
HRESULT ID3DXEffect::SetString(D3DXHANDLE hParameter, CONST LPCSTR pString);	Инициализирует идентифицируемую дескриптором **hParameter** строковую переменную в файле эффекта, копируя в нее текст из строки на которую указывает **pString**

Функция	Описание
HRESULT ID3DXEffect::SetTexture(**D3DXHANDLE hParameter,** **LPDIRECT3DBASETEXTURE9 pTexture** **);**	Инициализирует идентифицируемый дескриптором **hParameter** объект текстуры в файле эффекта на основании текстуры, на которую указывает **pTexture**
HRESULT ID3DXEffect::SetVector(**D3DXHANDLE hParameter,** **CONST D3DXVECTOR4* pVector** **);**	Инициализирует идентифицируемый дескриптором **hParameter** вектор в файле эффекта, копируя в него значения из вектора на который указывает **pVector**
HRESULT ID3DXEffect::SetVertexShader(**D3DXHANDLE hParameter,** **LPDIRECT3DVERTEXSHADER9 pVertexShader** **);**	Инициализирует идентифицируемый дескриптором **hParameter** объект вершинного шейдера в файле эффекта на основании вершинного шейдера, на который указывает **pVertexShader**
HRESULT ID3DXEffect::SetPixelShader(**D3DXHANDLE hParameter,** **LPDIRECT3DPIXELSHADER9 pPShader** **);**	Инициализирует идентифицируемый дескриптором **hParameter** объект пиксельного шейдера в файле эффекта на основании пиксельного шейдера, на который указывает **pPShader**

Мы получаем дескрипторы переменных (также называемых параметры эффекта) с помощью следующего метода:

```
D3DXHANDLE ID3DXEffect::GetParameterByName(
     D3DXHANDLE hParent, // область видимости переменной -
                         // родительская структура
     LPCSTR pName        // имя переменной
);
```

Его сигнатура аналогична методу **ID3DXConstantTable::GetConstantByName**. То есть первый параметр — это значение типа **D3DXHANDLE**, идентифицирующее родительскую структуру в пределах которой живет переменная, дескриптор которой мы хотим получить. Для глобальных переменных родительская структура отсутствует, и в этом параметре мы передаем **null**. Второй параметр — это имя переменной в том виде, в котором оно приведено в коде файла эффекта.

Для примера давайте взглянем на инициализацию нескольких переменных в файле эффекта:

```
// Данные для инициализации
D3DXMATRIX M;
D3DXMatrixIdentity(&M);

D3DXVECTOR4 color(1.0f, 0.0f, 1.0f, 1.0f);
```

```
IDirect3DTexture9* tex = 0;
D3DXCreateTextureFromFile(Device, "shade.bmp", &tex);

// Получаем дескрипторы параметров
D3DXHANDLE MatrixHandle = Effect->GetParameterByName(0, "Matrix");
D3DXHANDLE MtrlHandle   = Effect->GetParameterByName(0, "Mtrl");
D3DXHANDLE TexHandle    = Effect->GetParameterByName(0, "Tex");

// Инициализируем параметры
Effect->SetMatrix(MatrixHandle, &M);
Effect->SetVector(MtrlHandle, &color);
Effect->SetTexture(TexHandle, tex);
```

ПРИМЕЧАНИЕ Для каждого метода **`ID3DXEffect::Set*`** есть соответствующий метод **`ID3DXEffect::Get*`**, позволяющий приложению получить значение переменной из файла эффекта. Например, для получения значений матрицы мы можем использовать функцию

```
HRESULT ID3DXEffect::GetMatrix(
     D3DXHANDLE hParameter,
     D3DXMATRIX* pMatrix
);
```

Чтобы увидеть полный список методов, обратитесь к документации DirectX SDK.

19.6 Использование эффекта

В данном разделе и его подразделах мы покажем как использовать созданный эффект. В приведенном ниже списке перечислены действия, необходимые для использования эффекта:

1. Получить дескриптор техники из файла эффектов, которая будет использоваться.
2. Активировать требуемую технику.
3. Начать исполнение активной техники.
4. Для каждого прохода визуализации из активной техники визуализировать необходимые объекты. Вспомните, что техника может состоять из нескольких проходов визуализации и мы должны визуализировать объекты по одному разу для каждого прохода.
5. Закончить исполнение активной техники.

19.6.1 Получение дескриптора эффекта

Первый этап использования техники — это получение значения **`D3DXHANDLE`** для данной техники. Дескриптор техники может быть получен с помощью следующего метода:

```
D3DXHANDLE ID3DXEffect::GetTechniqueByName(
     LPCSTR pName // Имя техники
);
```

ПРИМЕЧАНИЕ На практике в файле эффекта обычно содержится несколько техник, разработанных для различных типов оборудования. Таким образом, приложение обычно выполняет провекру возможностей системы для определения возможностей оборудования и выборе на основе данных проверки наилучшей техники. Взгляните на описание метода **`ID3DXEffect::ValidateTechnique`** в следующем разделе.

19.6.2 Активация эффекта

После получения дескриптора выбранной техники надо ее активировать. Это делает следующий метод:

```
HRESULT ID3DXEffect::SetTechnique(
     D3DXHANDLE hTechnique // Дескриптор устанавливаемой техники
);
```

ПРИМЕЧАНИЕ Перед тем, как активировать технику вы должны убедиться, что она соответствует установленному оборудованию. То есть вы должны гарантировать, что установленное оборудование поддерживает все возможности, используемые в данной технике. Для этого можно использовать следующий метод:

```
HRESULT ID3DXEffect::ValidateTechnique(
     D3DXHANDLE hTechnique // Дескриптор проверяемой
                           // техники
);
```

Вспомните, что в файле эффекта может быть несколько техник, каждая из которых пытается реализовать требуемый эффект, используя свой набор возможностей оборудования, в надежде на то, что хотя бы одна реализация техники сможет работать на компьютере пользователя. Поэтому для эффекта вам надо перебрать все доступные техники и проверить каждую из них с помощью метода **`ID3DXEffect::ValidateTechnique`**, чтобы убедиться какие техники работают, а какие — нет, и дальше действовать соотвественно результатам.

19.6.3 Начало эффекта

Для визуализации объектов с использованием эффекта мы должны поместить вызовы функций рисования между вызовами методов **`ID3DXEffect::Begin`** и **`ID3DXEffect::End`**. Эти функции включают и выключают эффект соответственно.

```
HRESULT ID3DXEffect::Begin(
     UINT* pPasses,
     DWORD Flags
);
```

- **`pPasses`** — Возвращает количество проходов в активной технике.

- **Flags** — Любой из следующих флагов:
  - **Zero** (0) — Приказывает эффекту сохранить текущее состояние устройства и состояние шейдеров и восстановить их после завершения эффекта (при вызове **ID3DXEffect::End**). Это очень полезно, поскольку файл эффекта может менять состояния и часто требуется вернуть состояния, которые были до начала обработки эффекта.
  - **D3DXFX_DONOTSAVESTATE** — Приказывает не сохранять и не восстанавливать состояния устройства (за исключением состояния шейдеров).
  - **D3DXFX_DONOTSAVESHADERSTATE** — Приказывает эффекту не сохранять и не восстанавливать состояния шейдеров.

19.6.4 Установка текущего прохода визуализации

Перед тем, как начать рисование объектов с использованием эффекта, нам необходимо указать используемый проход визуализации. Вспомните, что техника состоит из одного или нескольких проходов визуализации, каждый из которых включает свои состояния устройства, выборки и/или шейдеры, которые применяются в данном проходе. Проход визуализации устанавливается с помощью следующего метода:

```
HRESULT ID3DXEffect::Pass(
     UINT iPass // Идентифицирующий проход индекс
);
```

Проходы визуализации для техники нумеруются в порядке 0 ... $n-1$ для n проходов. Таким образом, мы можем перебрать все проходы визуализации с помощью простого цикла **for** в теле которого будем рисовать все необходимые объекты. Пример реализации такого подхода приведен в разделе 19.6.6.

19.6.5 Завершение эффекта

После того, как завершена визуализация всех объектов во всех проходах, мы завершаем работу эффекта вызовом метода **ID3DXEffect::End**:

```
HRESULT ID3DXEffect::End(VOID);
```

19.6.6 Пример

Приведенный ниже фрагмент кода иллюстрирует все пять этапов, необходимых для использования эффекта:

```
// Файл эффекта:
technique T0
{
     pass P0
     {

     ...

     }
}
====================================

// Исходный код приложения

// Получаем дескриптор техники.
D3DXHANDLE hTech = 0;
hTech = Effect->GetTechniqueByName("T0");

// Активируем технику
Effect->SetTechnique(hTech);

// Начинаем активную технику
UINT numPasses = 0;
Effect->Begin(&numPasses, 0);

// Для каждого прохода визуализации
for(int i = 0; i < numPasses; i++)
{
     // Устанавливаем текущий проход
     Effect->Pass(i);

     // Визуализируем объекты для i-ого прохода
     Sphere->Draw();
}
// Завершаем эффект
Effect->End();
```

19.7 Пример приложения: освещение и текстурирование в файле эффектов

Для разминки давайте создадим файл эффекта, который выполняет освещение и текстурирование трехмерной модели. Пример в работе использует только функции фиксированного конвейера, показывая что каркас эффектов не ограничен эффектами, использующими шейдеры. На рис. 19.1 показано окно примера Lighting and Texturing.

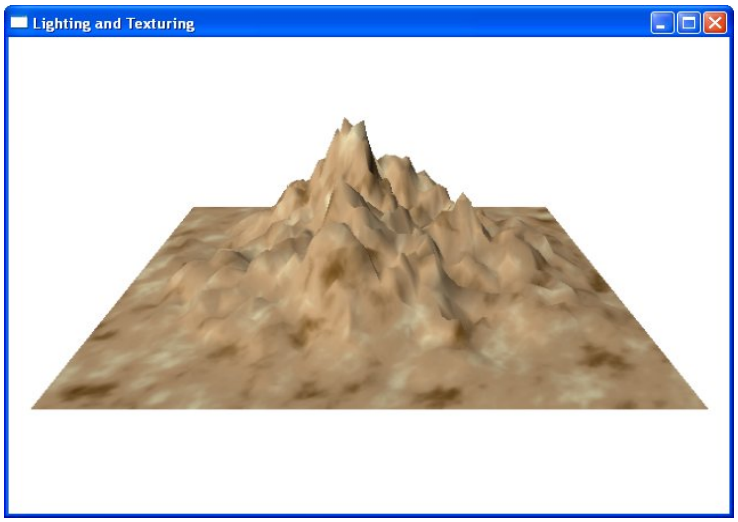

Рис. 19.1. *Окно приложения Lighting and Texturing. Текстуры, материал и режимы освещения заданы в файле эффекта*

Файл эффекта выглядит следующим образом:

```
//
// Файл:     light_tex.txt
// Описание: Файл эффекта, устанавливающий состояния устройства
//           для освещения и текстурирования трехмерной модели
//

//
// Глобальные переменные
//

matrix WorldMatrix;
matrix ViewMatrix;
matrix ProjMatrix;

texture Tex;

//
// Выборка
//

// Связываем текстуру Tex с соответствующим этапом текстурирования S0
// и задаем режимы выборки для этапа выборки S0
sampler S0 = sampler_state
{
     Texture   = (Tex);
     MinFilter = LINEAR;
     MagFilter = LINEAR;
     MipFilter = LINEAR;
};
```

```
//
// Эффект
//

technique LightAndTexture
{
     pass P0
     {
          //
          // Устанавливаем разные режимы визуализации
          //
          pixelshader      = null;   // Пиксельных шейдеров нет
          vertexshader     = null;   // Вершинных шейдеров нет
          fvf = XYZ | Normal | Tex1; // Настраиваемый формат вершин
          Lighting         = true;   // Разрешаем освещение
          NormalizeNormals = true;   // Нормализуем нормали
          SpecularEnable   = false;  // Отключаем отражаемый свет

          //
          // Установка состояний преобразования
          //
          WorldTransform[0]   = (WorldMatrix);
          ViewTransform       = (ViewMatrix);
          ProjectionTransform = (ProjMatrix);

          //
          // Инициализируем источник освещения с индексом 0.
          // Мы заполняем все компоненты light[0] потому что
          // в документации Direct3D рекомендуется заполнять все
          // компоненты для повышения производительности
          //
          LightType[0]         = Directional;
          LightAmbient[0]      = {0.2f, 0.2f, 0.2f, 1.0f};
          LightDiffuse[0]      = {1.0f, 1.0f, 1.0f, 1.0f};
          LightSpecular[0]     = {0.0f, 0.0f, 0.0f, 1.0f};
          LightDirection[0]    = {1.0f, -1.0f, 1.0f, 0.0f};
          LightPosition[0]     = {0.0f, 0.0f, 0.0f, 0.0f};
          LightFalloff[0]      = 0.0f;
          LightRange[0]        = 0.0f;
          LightTheta[0]        = 0.0f;
          LightPhi[0]          = 0.0f;
          LightAttenuation0[0] = 1.0f;
          LightAttenuation1[0] = 0.0f;
          LightAttenuation2[0] = 0.0f;

          // Разрешаем использовать этот источник света

          LightEnable[0] = true;

          //
          // Устанавливаем компоненты материала. Это аналогично
          // вызову IDirect3DDevice9::SetMaterial.
          //
          MaterialAmbient  = {1.0f, 1.0f, 1.0f, 1.0f};
```

```
            MaterialDiffuse  = {1.0f, 1.0f, 1.0f, 1.0f};
            MaterialEmissive = {0.0f, 0.0f, 0.0f, 0.0f};
            MaterialPower    = 1.0f;
            MaterialSpecular = {1.0f, 1.0f, 1.0f, 1.0f};

            //
            // Привязываем объект выборки S0 к
            // этапу выборки 0, который задается Sampler[0].
            //
            Sampler[0] = (S0);
      }
}
```

В этом файле эффекта мы сперва устанавливаем состояния устройства, как было описано в разделе 19.3. Например, непосредственно в файле эффекта мы устанавливаем источник света и материал. Кроме того, мы задаем матрицы преобразования, текстуру и режимы выборки. Эти состояния будут применены для любых объектов, которые визуализируются с использованием техники **`LightAndTexture`** в проходе визуализации **`P0`**.

ПРИМЕЧАНИЕ Обратите внимание, что ссылаясь на переменные из файла эффекта мы заключаем их имена в скобки. Например, для ссылок на матрицы преобразований мы должны писать **`(WorldMatrix)`**, **`(ViewMatrix)`** и **`(ProjMatrix)`**. Отсутствие скобок недопустимо.

Поскольку большая часть подготовительной работы, такая как установка освещения, материалов и текстур, выполняется в файле эффекта, в коде приложения достаточно создать эффект и разрешить его использование. В примере объявлены следующие глобальные переменные, относящиеся к рассматриваемой теме:

```
ID3DXEffect* LightTexEffect = 0;

D3DXHANDLE WorldMatrixHandle = 0;
D3DXHANDLE ViewMatrixHandle = 0;
D3DXHANDLE ProjMatrixHandle = 0;
D3DXHANDLE TexHandle = 0;

D3DXHANDLE LightTexTechHandle = 0;
```

Здесь нет ничего интересного — только указатель на **`ID3DXEffect`** и несколько дескрипторов. **`LightTexTechHandle`** — это дескриптор техники, на что указывает строка «Tech» в его имени.

Функция **`Setup`** выполняет следующие три действия: создает эффект, получает дескрипторы параметров эффекта и дескриптор той техники, которую мы будем использовать, и инициализирует некоторые из параметров эффекта. Вот код тех фрагментов функции, которые относятся к рассматриваемой теме:

```
bool Setup()
{
     HRESULT hr = 0;
```

```
//
// ...[Пропущена загрузка X-файла]
//

//
// Создание эффекта
//

ID3DXBuffer* errorBuffer = 0;
hr = D3DXCreateEffectFromFile(
      Device,           // связанное устройство
      "light_tex.txt",  // имя файла эффекта
      0,                // нет определений препроцессора
      0,                // нет интерфейса ID3DXInclude
      D3DXSHADER_DEBUG, // флаги компиляции
      0,                // нет совместного использования
                        // параметров
      &LightTexEffect,  // возвращает указатель на
                        // интерфейс эффекта
      &errorBuffer);    // возвращает сообщения об ошибках

// Выводим любые сообщения об ошибках
if(errorBuffer)
{
     ::MessageBox(0, (char*)errorBuffer->GetBufferPointer(),
                  0, 0);
     d3d::Release<ID3DXBuffer*>(errorBuffer);
}

if(FAILED(hr))
{
     ::MessageBox(0, "D3DXCreateEffectFromFile() - FAILED",
                  0, 0);
     return false;
}

//
// Сохраняем дескрипторы часто используемых параметров
//

WorldMatrixHandle = LightTexEffect->GetParameterByName(0,
                                              "WorldMatrix");
ViewMatrixHandle  = LightTexEffect->GetParameterByName(0,
                                              "ViewMatrix");
ProjMatrixHandle  = LightTexEffect->GetParameterByName(0,
                                              "ProjMatrix");
TexHandle         = LightTexEffect->GetParameterByName(0,
                                                  "Tex");
LightTexTechHandle =
      LightTexEffect->GetTechniqueByName("LightAndTexture");

//
// Устанавливаем параметры эффекта
//
```

```
     // Матрицы
     D3DXMATRIX W, P;

     D3DXMatrixIdentity(&W);
     LightTexEffect->SetMatrix(WorldMatrixHandle, &W);

     D3DXMatrixPerspectiveFovLH(
          &P, D3DX_PI * 0.25f, // 45 градусов
          (float)Width / (float)Height,
           1.0f, 1000.0f);

     LightTexEffect->SetMatrix(ProjMatrixHandle, &P);

     // Текстура
     IDirect3DTexture9* tex = 0;
     D3DXCreateTextureFromFile(Device,
                               "Terrain_3x_diffcol.jpg",
                               &tex);

     LightTexEffect->SetTexture(TexHandle, tex);

     d3d::Release<IDirect3DTexture9*>(tex);

     return true;
}
```

Функция **`Display`** прямолинейна и выполняет действия, описанные в разделе 19.6:

```
bool Display(float timeDelta)
{
     if(Device)
     {
          //
          // ...[Пропущено обновление камеры]
          //

          // Устанавливаем обновленную матрицу вида
          LightTexEffect->SetMatrix(ViewMatrixHandle, &V);

          //
          // Активируем технику и выполняем визуализацию
          //

          Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                        0xffffffff, 1.0f, 0);
          Device->BeginScene();

          // Устанавливаем используемую технику
          LightTexEffect->SetTechnique(LightTexTechHandle);

          UINT numPasses = 0;
          LightTexEffect->Begin(&numPasses, 0);
```

```
            for(int i = 0; i < numPasses; i++)
            {
                 LightTexEffect->Pass(i);

                 for(int j = 0; j < Mtrls.size(); j++)
                 {
                      Mesh->DrawSubset(j);
                 }
            }
            LightTexEffect->End();

            Device->EndScene();
            Device->Present(0, 0, 0, 0);
       }
       return true;
}
```

19.8 Пример приложения: туман

Одна из тем, которым мы к сожалению не можем посвятить целую главу — туман в Direct3D. Эффект тумана добавляет сцене новый уровень реализма и может использоваться для имитации различных погодных условий. Кроме того, туман может скрыть визуальные артефакты, возникающие на дальнем плане.

Хотя мы и не можем уделить этой теме то внимание, которого она заслуживает, здесь мы приведем краткий пример реализации тумана. Мы не будем вдаваться в детали, но покажем и исследуем код Direct3D, который является интуитивно понятным.

Туман в Direct3D является частью фиксированного конвейера функций и управляется через режимы визуализации. Приведенный ниже файл эффекта устанавливает все необходимые режимы для вершинного тумана.

ПРИМЕЧАНИЕ Direct3D также поддерживает пиксельный туман (также называемый табличным туманом), который является более точным, чем вершинный туман.

```
//
// Файл:      fog.txt
// Описание: Файл эффекта, устанавливающий режимы визуализации
//            для линейного вершинного тумана
//
technique Fog
{
     pass P0
     {
          //
          // Устанавливаем различные режимы визуализации
          //

          pixelshader      = null;
```

```
            vertexshader     = null;
            fvf              = XYZ | Normal;
            Lighting         = true;
            NormalizeNormals = true;
            SpecularEnable   = false;

            //
            // Режимы тумана
            //

            FogVertexMode = LINEAR; // Линейная функция тумана
            FogStart = 50.0f;       // Туман начинается в
                                    // 50 единицах от камеры.
            FogEnd = 300.0f;        // Туман заканчивается в
                                    // 300 единицах от камеры
            FogColor = 0x00CCCCCC;  // Туман серого цвета
            FogEnable = true;       // Разрешить вершинный туман
        }
}
```

Как видите, линейный вершинный туман управляется через пять простых режимов визуализации:

- **`FogVertexMode`** — Задает функцию тумана, которая будет использоваться для вершинного тумана. Функция тумана определяет как изменяется плотность тумана с увеличением расстояния до камеры, поскольку естественно, что туман менее плотный возле камеры и становится более плотным по мере увеличения расстояния. Можно использовать значения **`LINEAR`**, **`EXP`** и **`EXP2`**. Эти функции определены следующим образом:

  Функция тумана **`LINEAR`**: $$f = \frac{end - d}{end - start}$$

  Функция тумана **`EXP`**: $$f = \frac{1}{e^{d \cdot density}}$$

  Функция тумана **`EXP2`**: $$f = \frac{1}{e^{(d \cdot density)^2}}$$

  (d — это расстояние до камеры.)

ПРИМЕЧАНИЕ	Если вы используете функции тумана **`EXP`** или **`EXP2`**, то вам не надо задавать значения **`FogStart`** и **`FogEnd`**, поскольку в этих типах функций они не используются; вместо этого вы должны задать режим визуализации для плотности тумана (т.е. **`FogDensity = какое-нибудь число;`**).

- **`FogStart`** — Отмечает стартовую глубину, начиная с которой объекты будут затуманиваться.
- **`FogEnd`** — Отмечает конечную глубину, после которой затуманивание объектов прекращается.

ПРИМЕЧАНИЕ	`FogStart` и `FogEnd` определяют интервал значений глубины (от камеры) в котором объекты будут затуманиваться.

- **`FogColor`** — Значение типа **`DWORD`** или **`D3DCOLOR`**, задающее цвет тумана.
- **`FogEnable`** — Укажите **`true`**, чтобы включить вершинный туман или **`false`**, чтобы запретить вершинный туман.

К любым объектам, которые визуализируются с использованием файла эффекта fog.txt будет применен эффект тумана. Благодаря этому мы можем контроллировать, какие объекты будут затуманены, а какие — нет. Это полезно для выборочного затенения областей. Например, обычно туман находится на улице, а внутри домов тумана нет. Также часто требуется скрывать в тумане только определенные части территории, а другие оставлять незатуманенными. На рис. 19.2 показано окно рассматриваемого в данном разделе примера Fog Effect.

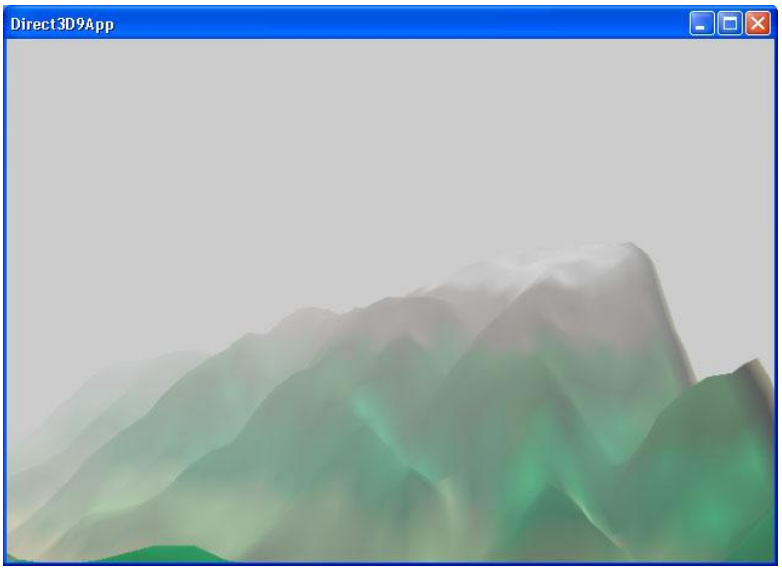

Рис. 19.2. *Окно программы Fog Effect. В этом примере мы используем линейную функцию тумана и режимы визуализации тумана задаются в файле эффекта*

19.9 Пример приложения: мультипликационный эффект

Оба рассмотренных нами к настоящему моменту файла эффектов не используют шейдеры. Поскольку шейдеры обычно являются важной составной частью эффектов, стоит рассмотреть хотя бы один пример их совместного использования. Пример CartoonEffect реализует мультипликационное затенение, которое обсуждалось в главе 17, но на этот раз мы воспользуемся каркасом эффектов. Посмотрите на слегка урезанную версию файла эффекта:

```
//
// Файл:     tooneffect.txt
// Описание: Мультипликационное затенение в файле эффектов.
//

extern matrix WorldMatrix;
extern matrix ViewMatrix;
extern matrix ProjMatrix;
extern vector Color;
extern vector LightDirection;
static vector Black = {0.0f, 0.0f, 0.0f, 0.0f};
extern texture ShadeTex;

struct VS_INPUT
{
     vector position : POSITION;
     vector normal : NORMAL;
};

struct VS_OUTPUT
{
     vector position : POSITION;
     float2 uvCoords : TEXCOORD;
     vector diffuse : COLOR;
};

// Шейдер мультипликационного затенения
VS_OUTPUT Main(VS_INPUT input)
{
     ...[Код шейдера для краткости пропущен]
}

sampler ShadeSampler = sampler_state
{
     Texture = (ShadeTex);
     MinFilter = POINT; // фильтрация в мультипликационном
                        // затенении отключается
     MagFilter = POINT;
     MipFilter = NONE;
};
```

```
technique Toon
{
      pass P0
      {
            // Устанавливаем вершинный шейдер прохода P0
            vertexShader = compile vs_1_1 Main();

            // Связываем объект выборки с этапом выборки 0.
            Sampler[0] = (ShadeSampler);
      }
}
```

Обратите внимание, что функция шейдера мультипликационного затенения определена в файле эффекта, и мы указываем, какой шейдер будет применяться в данном проходе, с помощью синтаксиса **`vertexShader = compile vs_1_1 Main();`** в блоке прохода. Состояния устройства как обычно устанавливаются в файле эффекта.

19.10 EffectEdit

Перед тем, как завершить данную главу, упомянем о программе EffectEdit, которая поставляется вместе с DirectX SDK. Вы найдете ее в каталоге \DXSDK\Samples\C++\Direct3D\Bin. Окно этой программы показано на рис. 19.3.

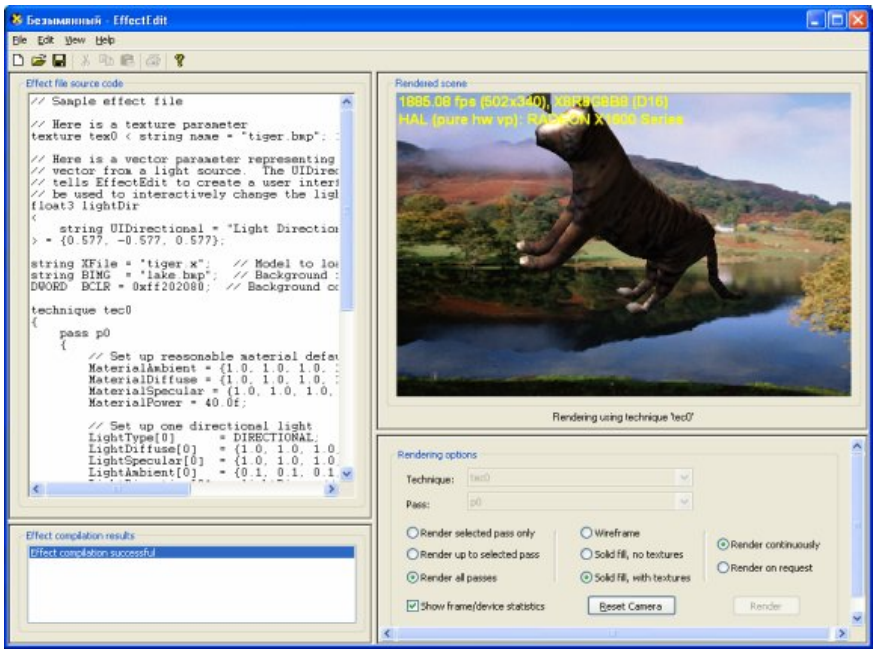

Рис. 19.3. *Окно программы EffectEdit, поставляемой вместе с DirectX SDK*

Программа EffectEdit полезна для написания и тестирования файлов эффектов. Мы рекомендуем вам выделить время для знакомства с этой утилитой.

19.11 Итоги

- Файл эффекта объединяет завершенную реализацию эффекта, включая варианты обработки аппаратных сбоев, вызванных различиями в поддерживаемых аппаратурой возможностях, и проходы визуализации. Каркас эффектов полезен тем, что позволяет нам вносить изменения в файл эффекта без перекомпиляции всего приложения, а также позволяет объединить весь относящийся к эффекту код в одном файле, что повышает модульность. Файлы эффектов могут использоваться без шейдеров; вы вполне можете создать файл эффекта, который будет использовать только функции фиксированного конвейера.
- Техникой называется отдельная реализация конкретного эффекта. Обычно файл эффекта состоит из нескольких техник, которые реализуют один и тот же эффект, но разными способами. Каждая реализация использует свой собственный набор необходимых ей возможностей видеокарты. Таким образом, приложение может выбрать технику, которая наиболее точно соотвствует возможностям установленного в компьютере оборудования. Например, реализуя мультитекстурирование, мы можем описать две техники — одну с использованием пиксельных шейдеров и другую, работающую только с функциями фиксированного конвейера. Тогда те пользователи, чьи видеокарты поддерживают пиксельные шейдеры могут воспользоваться преимуществами реализации эффекта с использованием шейдеров, а те пользователи, чьи видеокарты не поддерживают шейдеры, будут использовать реализацию эффекта, работающую с фиксированным конвейером.
- Техника состоит из одного или нескольких проходов визуализации. Проход визуализации состоит из состояний устройства и шейдеров, используемых для визуализации объектов в данном проходе. Несколько проходов визуализации необходимы потому что многие эффекты требуют, чтобы одни и теже объекты визуализировались несколько раз подряд с различными состояниями устройства и/или шейдерами.

Приложение
Введение в программирование для Windows

Чтобы использовать интерфейс программирования приложений (API) Direct3D необходимо создать приложение Windows (Win32-приложение) с главным окном в котором мы будем визуализировать наши трехмерные сцены. Данное приложение служит введением в написание приложений для Windows с использованием «чистого» Win32 API. В общих чертах Win32 API представляет собой набор низкоуровневых функций и структур, доступных из языка C и позволяющих нашему приложению и операционной системе Windows взаимодействовать друг с другом. Например, чтобы приказать Windows показать заданное окно, используется функция Win32 API **`ShowWindow`**.

Программирование для Windows — огромная тема, и это приложение знакомит только с теми моментами, которые необходимы при работе с Direct3D. Читатели, желающие больше узнать о программировании для Windows с использованием Win32 API, могут обратиться к ставшей классическим трудом по этой теме книге Чарльза Петзольда «Programming Windows» (к ее последнему, пятому изданию). Другим незаменимым ресурсом при работе с технологиями Microsoft является библиотека MSDN, которая обычно входит в Microsoft Visual Studio, но также доступна в Интернете по адресу *www.msdn.microsoft.com*. Вобщем, если вы встретите функцию или структуру Win32 о которой захотите узнать больше, откройте MSDN и выполните поиск по названию функции или структуры. В этом приложении мы достаточно часто будем отсылать вас за дополнительными сведениями о функциях и структурах к MSDN.

Цели

- Изучить управляемую событиями модель программирования и понять, как она используется в Windows.
- Изучить код минимального приложения Windows, достаточного для работы с Direct3D.

Обзор

Основной темой программирования для Windows, как следует из названия этой операционной системы, является программирование окон. Большинство компонентов Windows-приложений — главные окна, меню, панели инструментов, полосы прокрутки, кнопки и другие элементы управления — являются окнами. Следовательно, типичное Windows-приложение состоит из нескольких окон. В нескольких следующих подразделах приводится краткий обзор концепций программирования для Windows с которыми необходимо познакомиться перед тем, как перейти к более детальному обсуждению.

Ресурсы

В Windows могут одновременно работать несколько приложений. Следовательно, аппаратные ресурсы, такие как время процессора, память и даже экран монитора, совместно используются несколькими приложениями. Чтобы предотвратить хаос, который возникнет, если несколько программ одновременно попытаются получить доступ к ресурсам и изменить их состояние, Windows полностью запрещает приложениям прямой доступ к аппаратным средствам компьютера. Одна из главных задач Windows — управление запущенными программами и распределение между ними ресурсов. В результате, если нашему приложению надо сделать что-нибудь не оказывая влияния на другие выполняющиеся программы, оно должно делать это через механизмы Windows. Например, для того чтобы отобразить окно, вы должны вызвать функцию **`ShowWindow`**, а не записывать данные непосредственно в память видеокарты.

События, сообщения, очередь сообщений и цикл обработки сообщений

Приложения Windows следуют *управляемой событиями модели программирования* (*event-driving programming model*). Обычно приложение Windows просто сидит и ждет пока не произойдет какое-нибудь *событие* (*event*) (приложение может выполнять фоновую работу — то есть выполнять какие-то задачи, когда не происходит никаких событий). События генерируются во многих случаях; наиболее общие примеры — нажатие клавиш, щелчки мыши, создание, перемещение, сворачивание, развертывание и закрытие окон, изменение размеров и отображение окна.

Когда происходит событие Windows отправляет приложению *сообщение* (*message*), уведомляющее о событии, и помещает его в *очередь сообщений* (*message queue*) приложения, которая представляет собой обычную очередь, где хранятся поступившие приложению сообщения. Приложение постоянно проверяет состояние очереди в *цикле обработки сообщений* (*message loop*), и, когда обнаруживает в очереди новое сообщение, направляет его *оконной процедуре* (*window procedure*) того окна, которому данное сообщение предназначено. (Вспомните, что у приложения может быть несколько окон.)

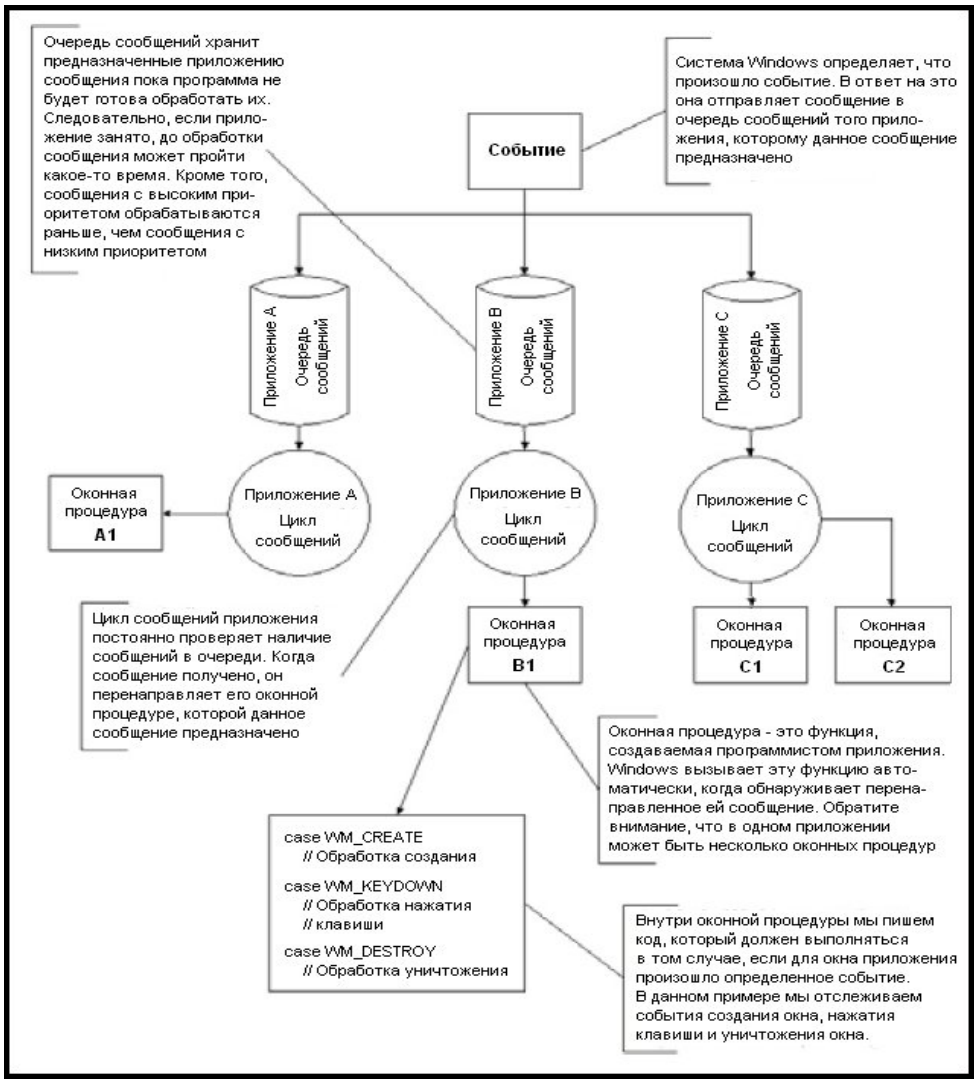

Рис. 1. *Управляемая событиями модель программирования*

Оконная процедура — это специальная функция, связанная с окном приложения. (У каждого окна должна быть оконная процедура, но несколько окон могут совместно использовать одну оконную процедуру. Следовательно нет необходимости писать для каждого окна отдельную оконную процедуру.) В оконной процедуре мы реализуем обработку различных сообщений. Например, мы можем реализовать завершение работы приложения при нажатии клавиши **Esc**. Для этого в оконной процедуре следует написать:

```
case WM_KEYDOWN:
     if( wParam == VK_ESCAPE )
          ::DestroyWindow(MainWindowHandle);
     return 0;
```

Сообщения, которые окно не обрабатывает обычно переправляются стандартной оконной процедуре, которая и занимается их обработкой.

Итак, подытожим. Пользователь или приложение выполняют какое-нибудь действие, приводящее к генерации события. Операционная система находит приложение, которому предназначено данное событие, и посылает ему сообщение. Отправленное сообщение добавляется к очереди сообщений приложения. Приложение постоянно проверяет свою очередь сообщений. Когда приложение обнаруживает в очереди предназначенное ему сообщение, оно направляет его окноной процедуре связанной с тем окном, которому предназначалось сообщение. После получения сообщения оконная процедура выполняет инструкции, отвечающие за обработку поученного сообщения. Все эти действия показаны на рис. 1.

GUI

Большинство Windows-приложениий предоставляют пользователю для работы графический интерфейс (GUI, graphical user interface). Обычное Windows-приложение содержит главное окно, меню, панель инструментов и, возможно, ряд других элементов управления. На рис. 2 показаны наиболее распространенные элементы графического интерфейса пользователя. Для игр, использующих Direct3D. нам не требуется профессиональный интерфейс пользователя. Фактически нам достатоно главного окна, в клиентской области которого мы будем отображать трехмерные сцены.

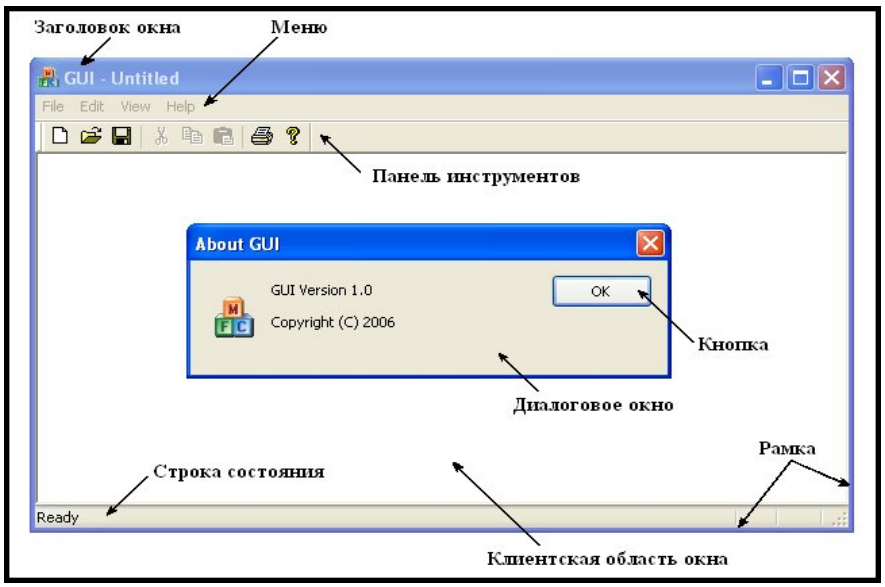

Рис. 2. *Графический интерфейс пользователя обычного приложения Windows. Клиентская область — это белое пространство в окне приложения. Обычно она используется для показа пользователю результатов работы программы. Создавая Direct3D-приложения мы используем эту область для визуализации наших трехмерных сцен*

Windows-приложение Hello World

Ниже приведен код полнофункциональной и очень простой программы для Windows. Лучше всего просто следовать за кодом. В следующих разделах мы исследуем его строка за строкой. Мы рекомендуем вам в качестве упражнения создать проект в вашей среде разработки, вручную набрать код, скомпилировать его и запустить. Обратите внимание, что выбирая тип проекта надо указать Win32 Application, а не Win32 Console Application.

```
//////////////////////////////////////////////////////////////////////
//
// Файл: hello.cpp
//
// Автор: Фрэнк Д. Луна (C) All Rights Reserved
//
// Система: AMD Athlon 1800+ XP, 512 DDR, Geforce 3, Windows XP,
// MSVC++ 7.0
//
// Описание: Демонстрация создания приложения для Windows.
//
//////////////////////////////////////////////////////////////////////

// Включение заголовочного файла, содержащего все объявления
// структур, типов данных и функций Win32 API необходимых для
// Windows-программы.
#include <windows.h>

// Дескриптор главного окна. Используется для идентификации
// главного окна, которое мы создадим
HWND MainWindowHandle = 0;

// Обертка для кода, необходимого для инициализации приложения
// Windows. Функция возвращает true если инициализация произведена
// успешно и false в ином случае.
bool InitWindowsApp(HINSTANCE instanceHandle, int show);

// Обертка для кода цикла сообщений.
int Run();

// Оконная процедура, обрабатывающая получаемые нашим окном
// сообщения
LRESULT CALLBACK WndProc(HWND hWnd,
                         UINT msg,
                         WPARAM wParam,
                         LPARAM lParam);

// Эквивалент main() для Windows
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   PSTR pCmdLine,
                   int nShowCmd)
{
```

```
     // Сперва мы создаем и инициализируем наше приложение Windows
     // Обратите внимание, что значения hInstance и nShowCmd
     // передаются WinMain в параметрах.

     if(!InitWindowsApp(hInstance, nShowCmd))
     {
          ::MessageBox(0, "Init - Failed", "Error", MB_OK);
          return 0;
     }

     // После создания и инициализации приложения мы входим
     // в цикл обработки сообщений. В нем мы остаемся до тех пор
     // пока не будет получено сообщение WM_QUIT, говорящее, что
     // работа приложения должна быть завершена.

     return Run(); // вход в цикл сообщений
}

bool InitWindowsApp(HINSTANCE instanceHandle, int show)
{
     // Первая задача при создании окна - описать его
     // характеристики путем заполнения структуры WNDCLASS

     WNDCLASS wc;

     wc.style         = CS_HREDRAW | CS_VREDRAW;
     wc.lpfnWndProc   = WndProc;
     wc.cbClsExtra    = 0;
     wc.cbWndExtra    = 0;
     wc.hInstance     = instanceHandle;
     wc.hIcon         = ::LoadIcon(0, IDI_APPLICATION);
     wc.hCursor       = ::LoadCursor(0, IDC_ARROW);
     wc.hbrBackground =
          static_cast<HBRUSH>(::GetStockObject(WHITE_BRUSH));
     wc.lpszMenuName  = 0;
     wc.lpszClassName = "Hello";

     // Затем мы регистрируем описание класса окна в Windows
     // чтобы потом мы смогли создать окно с объявленными
     // характеристиками

     if(!::RegisterClass(&wc))
     {
          ::MessageBox(0, "RegisterClass - Failed", 0, 0);
          return false;
     }


     // После регистрации описания нашего класса окна мы можем
     // создать окно с помощью функции CreateWindow.
     // Обратите внимание, что функция возвращает значение HWND
     // для созданного окна, которое мы сохраняем в переменной
     // MainWindowHandle. В дальнейшем переменная MainWindowHandle
     // позволит обращаться именно к тому окну, которое мы создали.
```

```
     MainWindowHandle = ::CreateWindow(
                                   "Hello",
                                   "Hello",
                                   WS_OVERLAPPEDWINDOW,
                                   CW_USEDEFAULT,
                                   CW_USEDEFAULT,
                                   CW_USEDEFAULT,
                                   CW_USEDEFAULT,
                                   0,
                                   0,
                                   instanceHandle,
                                   0);

     if(MainWindowHandle == 0)
     {
          ::MessageBox(0, "CreateWindow - Failed", 0, 0);
          return false;
     }

     // Теперь мы отображаем и обновляем только что созданное
     // окно. Обратите внимание, что в качестве параметра обоим
     // функциям передается значение MainWindowHandle, чтобы они
     // знали какое именно окно надо отображать и обновлять.
     ::ShowWindow(MainWindowHandle, show);
     ::UpdateWindow(MainWindowHandle);

     return true;
}

int Run()
{
     MSG msg;
     ::ZeroMemory(&msg, sizeof(MSG));

     // Цикл выполняется, пока мы не получим сообщение WM_QUIT.
     // Функция GetMessage возвращает 0 (false) только когда
     // получено сообщение WM_QUIT, что приводит к выходу из цикла.
     while(::GetMessage(&msg, 0, 0, 0))
     {
          // Трансляция сообщения и его перенаправление
          // соответствующей оконной процедуре.
          ::TranslateMessage(&msg);
          ::DispatchMessage(&msg);
     }

     return msg.wParam;
}

LRESULT CALLBACK WndProc(HWND windowHandle,
                         UINT msg,
                         WPARAM wParam,
                         LPARAM lParam)
{
     // Обработка заданных сообщений:
     switch( msg )
     {
```

```
      case WM_LBUTTONDOWN:
           // Если нажата левая кнопка мыши,
           // отображаем диалоговое окно.
           ::MessageBox(0, "Hello, World", "Hello", MB_OK);
           return 0;

      case WM_KEYDOWN:
           // Если нажата клавиша Esc, уничтожаем
           // главное окно приложения, идентифицируемое
           // дескриптором MainWindowHandle.
           if( wParam == VK_ESCAPE )
                ::DestroyWindow(MainWindowHandle);
           return 0;

      case WM_DESTROY:
           // Если получено сообщение о завершении работы,
           // отправляем сообщение, которое завершит работу
           // цикла обработки сообщений.
           ::PostQuitMessage(0);
           return 0;
      }

      // Переправляем все остальные сообщения, которые
      // наша программа не обрабатывает сама, системной
      // процедуре обработки сообщений.
      return ::DefWindowProc(windowHandle,
                             msg,
                             wParam,
                             lParam);
}
```

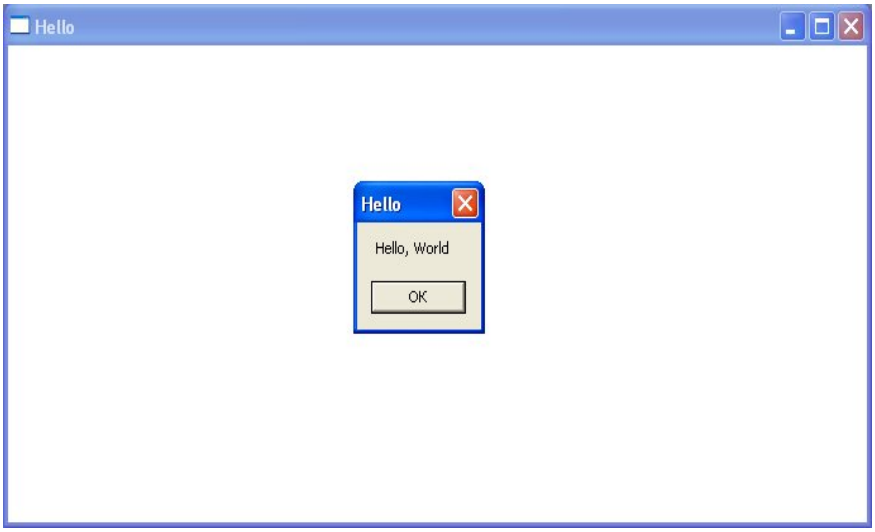

Рис. 3. *Окно приведенной выше программы. Обратите внимание, что окно сообщений появляется, если вы нажмете левую кнопку мыши, когда указатель находится в клиентской области окна*

Исследование программы Hello World

Давайте исследуем приведенный код сверху донизу, уделяя внимание каждому встретившемуся вызову функции. Читая следующие подразделы, смотрите на приведенный выше код программы Hello World.

Включение файлов, глобальные переменные и прототипы

Первое, что мы должны сделать — включить заголовочный файл windows.h. Включив этот файл мы получаем доступ к объявлениям структур, типов и функций, необходимых для использования базовых элементов Win32 API.

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

Вторая инструкция — это объявление глобальной переменной с типом **HWND**. Это сокращение обозначает «дескриптор окна» (handle to a window). Программируя для Windows вы часто будете пользоваться дескрипторами для ссылок на внутренние объекты Windows. В данном примере мы используем **HWND** чтобы ссылаться на главное окно приложения, управление которым осуществляют внутренние механизмы Windows. Мы должны сохранить дескриптор нашего окна потому что многие вызовы API требуют, чтобы им был передан дескриптор того окна, над которым они должны произвести действия. Например, функция **UpdateWindow** получает один параметр типа **HWND**, используемый для того, чтобы сообщить ей, какое окно должно быть обновлено. Если мы не передадим дескриптор, функция не будет знать какое окно ей обновлять.

```
HWND MainWindowHandle = 0;
```

В следующих трех строках находятся объявления функций. Говоря кратко, **InitWindowsApp** создает и инициализирует главное окно приложения, **Run** является оберткой для цикла обработки сообщений нашего приложения, а **WndProc** — это оконная процедура главного окна нашей программы. Мы подробно исследуем эти функции, когда дойдем до того места кода, где они вызываются.

```
bool InitWindowsApp(HINSTANCE instanceHandle, int show);
int Run();
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
```

WinMain

WinMain в мире Windows является аналогом функции **main** в обычном программировании на C++. Прототип **WinMain** выглядит так:

```
int WINAPI WinMain(
     HINSTANCE hInstance,
     HINSTANCE hPrevInstance,
     LPSTR     lpCmdLine,
     int       nCmdShow
);
```

- **hInstance** — Дескриптор экземпляра данного приложения. Он предназначен для идентификации конкретного приложения и ссылок на него. Помните, что в Windows могут одновременно работать несколько приложений и поэтому необходим механизм, позволяющий идентифицировать каждое из них.
- **hPrevInstance** — В 32-разрядных системах не используется и равно 0.
- **lpCmdLine** — Строка с аргументами командной строки, указанными при запуске программы.
- **nCmdShow** — Вариант отображения окна приложения. Наиболее часто используются варианты **SW_SHOW** (отображение окна указанных размеров в заданной позиции), **SW_SHOWMAXIMIZED** (отображение окна, развернутого на весь экран) и **SW_SHOWMINIMIZED** (отображение свернутого окна). Полный список вариантов отображения и соответствующих констант приведен в библиотеке MSDN.

Если работа функции **WinMain** завершается успешно, она должна вернуть член **wParam** сообщения **WM_QUIT**. Если работа функции завершена до входа в цикл обработки сообщений, она должна вернуть 0. Идентификатор **WINAPI** определен следующим образом:

```
#define WINAPI __stdcall
```

Он задает правила вызова функций и определяет, как функция будет обращаться к размещенным в стеке параметрам.

ПРИМЕЧАНИЕ В прототипе функции **WinMain** в примере Hello World, для третьего параметра мы указали тип **PSTR**, а не **LPSTR**. Это объясняется тем, что в 32-разрядных системах Windows больше нет «дальних указателей». **PSTR** — это просто указатель на строку символов (т.е., **char***).

WNDCLASS и регистрация

Из **WinMain** мы обращаемся к функции **InitWindowsApp**. Как уже сообщалось, эта функция выполняет действия, необходимые для инициализации программы. Давайте перейдем к ее коду и исследуем его. **InitWindowsApp** возвращает **true** или **false** — **true**, если инициализация успешно завершена, **false** если что-то не получилось. Как видно из кода **WinMain**, мы передаем функции **InitWindowsApp** копию дескриптора экземпляра нашего приложения

и переменную, задающую режим отображения окна. Сама функция **WinMain** получает эти два значения в своих параметрах.

```
if(!InitWindowsApp(hInstance, nShowCmd))
```

Первая задача, с которой мы сталкиваемся при инициализации окна, — описать параметры окна и зарегистрировать его в Windows. Параметры окна задаются с помощью структуры данных **WNDCLASS**. Вот ее определение:

```
typedef struct _WNDCLASS {
     UINT    style;
     WNDPROC lpfnWndProc;
     int     cbClsExtra;
     int     cbWndExtra;
     HANDLE  hInstance;
     HICON   hIcon;
     HCURSOR hCursor;
     HBRUSH  hbrBackground;
     LPCTSTR lpszMenuName;
     LPCTSTR lpszClassName;
} WNDCLASS;
```

- **style** — Задает стиль окна. В нашем примере мы используем комбинацию флагов **CS_HREDRAW** и **CS_VREDRAW**. Она означает, что окно будет перерисовываться при изменении его размеров по горизонтали или по вертикали. Полный список стилей с их описанием приведен в библиотеке MSDN.

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

- **lpfnWndProc** — Указатель на оконную процедуру. Именно здесь устанавливается связь оконной процедуры с окном. Таким образом окна, созданные на основе одного и того же экземпляра структуры **WNDCLASS** будут совместно использовать одну и ту же оконную процедуру. Сама оконная процедура будет рассмотрена чуть позже в разделе «Оконная процедура».

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

- **cbClsExtra** и **cbWndExtra** — Это дополнительные области памяти, которые вы можете использовать в своих собственных целях. В рассматриваемой программе дополнительные области памяти не нужны и поэтому обоим параметрам присваивается 0.

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

- **hInstance** — Поле для дескриптора экземпляра нашего приложения. Вспомните, что этот дескриптор был передан нам через функцию **WinMain**.

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

- **hIcon** — Дескриптор значка, используемого для окон, создаваемых на основе данного класса. Существует несколько стандартных значков операционной системы и вы можете выбрать один из них. Более подробно этот вопрос рассматривается в MSDN.

```
wc.hIcon = ::LoadIcon(0, IDI_APPLICATION);
```

- **hCursor** — Тут, аналогично полю **hIcon**, вы задаете дескриптор курсора приложения, определяющий как будет выглядеть указатель мыши, когда он находится в клиентской области окна. Здесь также есть несколько встроенных типов курсоров. За дополнительной информацией обращайтесь к MSDN.

```
wc.hCursor = ::LoadCursor(0, IDC_ARROW);
```

- **hbrBackground** — Это поле определяет цвет фона клиентской области окна. В нашем примере мы вызываем функцию **GetStockObject**, которая возвращает дескриптор кисти указанного нами цвета. Описание других встроенных кистей можно найти в MSDN.

```
wc.hbrBackground =
     static_cast<HBRUSH>(::GetStockObject(WHITE_BRUSH));
```

- **lpszMenuName** — Задает меню окна. В нашем приложении нет меню, поэтому значение данного поля равно 0.

```
wc.lpszMenuName = 0;
```

- **lpszClassName** — Определяет имя создаваемого класса окна. Вы можете выбрать любое имя. В нашем приложении класс называется «Hello». Имя используется для идентификации структуры данных класса, чтобы мы могли обращаться к ней в дальнейшем.

```
wc.lpszClassName = "Hello";
```

После того, как мы описали параметры класса нашего окна, нам надо зарегистрировать его в Windows. Это выполняется с помощью функции **RegisterClass**, которая получает указатель на структуру **WNDCLASS**. В случае успешного завершения функция возвращает 0.

```
if(!::RegisterClass(&wc))
```

Создание и отображение окна

После того, как мы зарегистрировали в Windows переменную **WNDCLASS**, можно на основании содержащегося в ней описания класса создать новое окно. Для ссылки на структуру **WNDCLASS**, описывающую окно, которое мы хотим создать, используется заданное в члене **lpszClassName** имя класса. Для создания окна мы используем функцию **CreateWindow**, прототип которой выглядит следующим образом:

```
HWND CreateWindow(
     LPCTSTR lpClassName,
     LPCTSTR lpWindowName,
     DWORD dwStyle,
     int x,
     int y,
     int nWidth,
     int nHeight,
     HWND hWndParent,
     HMENU hMenu,
     HANDLE hInstance,
     LPVOID lpParam
);
```

- **lpClassName** — Имя класса (завершающаяся нулем строка), которое было указано в зарегистрированной структуре **WNDCLASS**, описывающей параметры окна, которое мы хотим создать. Передавайте имя класса, указанное в той структуре **WNDCLASS**, которую вы хотите использовать при создании окна.
- **lpWindowName** — Имя (завершающаяся нулем строка), которое мы присваиваем нашему окну; это имя будет также отображаться в заголовке окна.
- **dwStyle** — Описывает стиль создаваемого окна. Используемое в примере Hello World значение **WS_OVERLAPPEDWINDOW** является комбинацией флагов **WS_OVERLAPPED**, **WS_CAPTION**, **WS_SYSMENU**, **WS_THICKFRAME**, **WS_MINIMIZEBOX** и **WS_MAXIMIZEBOX**. Имена флагов описывают соответствующие характеристики окна. Полный список стилей приведен в MSDN.
- **x** — Позиция по горизонтали верхнего левого угла окна в экранной системе координат.
- **y** — Позиция по вертикали верхнего левого угла окна в экранной системе координат.
- **nWidth** — Ширина окна в пикселях.
- **nHeight** — Высота окна в пикселях.
- **hWndParent** — Дескриптор окна, которое является родителем данного. Создаваемое в примере окно не имеет взаимоотношений с другими окнами, поэтому данному параметру присваивается 0.
- **hMenu** — Дескриптор меню. В приложении Hello World нет меню, поэтому данному параметру присваивается 0.
- **hInstance** — Дескриптор экземпляра приложения с которым связано данное окно.
- **lpParam** — указатель на определяемые пользователем данные.

ПРИМЕЧАНИЕ Когда мы указываем координаты окна (*x*, *y*), они отсчитываются относительно верхнего левого угла экрана. При этом положительные значения по оси X отсчитываются, как обычно, вправо, а вот положительные значения по оси Y отсчитываются вниз. Эта система координат, называемая *экранными координатами* (*screen coordinates*) или *экранным пространством* (*screen space*), показана на рис. 4.

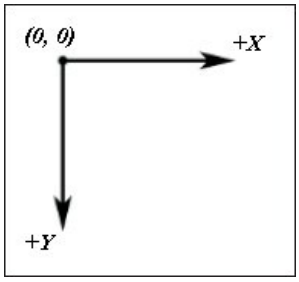

Рис. 4. *Экранное пространство*

Функция **CreateWindow** возвращает дескриптор созданного ею окна (значение типа **HWND**). Если создать окно не удалось, значение дескриптора равно нулю. Помните, что дескриптор — это способ сослаться на конкретное окно, управляемое Windows. Многие вызовы API требуют указания **HWND**, чтобы знать с каким окном производятся действия.

Последние два обращения к функциям API из функции **InitWindowsApp** предназначены для отображения окна. Сперва мы вызываем функцию **ShowWindow** и передаем ей дескриптор только что созданного окна, чтобы Windows знала, какое окно должно быть показано. Мы также передаем число, определяющее в каком виде будет показано окно (обычным, свернутым, развернутым на весь экран и т.д.). Здесь лучше всего указать значение **nShowCmd**, которое было передано нам в одном из аргументов **WinMain**. Конечно, вы можете жестко задать значение в коде, но это не рекомендуется. После отображения окна мы должны обновить его. Это делает функция **UpdateWindow**; она получает один аргумент, являющийся дескриптором обновляемого окна.

```
::ShowWindow(MainWindowHandle, show);
::UpdateWindow(MainWindowHandle);
```

Если в функции **InitWindowsApp** были выполнены все описанные выше действия, значит инициализация завершена; мы возвращаем **true**, чтобы сообщить об успешном завершении функции.

Цикл сообщений

Успешно завершив инициализацию, мы переходим к сердцу программы — циклу обработки сообщений. В программе Hello World, мы заключили цикл обработки сообщений в функцию с именем **Run**.

```
int Run()
{
     MSG msg;
     ::ZeroMemory(&msg, sizeof(MSG));

     while(::GetMessage(&msg, 0, 0, 0))
     {
          ::TranslateMessage(&msg);
          ::DispatchMessage(&msg);
     }
     return msg.wParam;
}
```

Начинается функция **Run** с объявления переменной **msg** типа **MSG**, являющейся структурой данных, представляющей сообщения Windows. Ее определение выглядит следующим образом:

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

- **hwnd** — Идентифицирует окно, которому предназначено сообщение.
- **message** — Одна из предопределенных констант, идентифицирующих сообщение (например, **WM_QUIT**).
- **wParam** — Дополнительная информация о сообщении. Зависит от конкретного сообщения.
- **lParam** — Дополнительная информация о сообщении. Зависит от конкретного сообщения.
- **time** — Время, когда сообщение было помещено в очередь.
- **pt** — Координаты указателя мыши (x, y) в экранном пространстве в тот момент, когда сообщение было помещено в очередь.

Затем мы входим в цикл обработки сообщений. Функция **GetMessage** будет возвращать **true** до тех пор, пока из очереди не будет извлечено сообщение **WM_QUIT**; следовательно цикл будет выполняться, пока не будет получено сообщение **WM_QUIT**. Функция **GetMessage** извлекает сообщение из очереди и заполняет члены нашей структуры **MSG**. Если **GetMessage** возвращает **true**, будут вызваны еще две функции: **TranslateMessage** и **DispatchMessage**. **TranslateMessage** выполняет трансляцию сообщений Windows, в частности преобразование виртуальных кодов клавиш в коды символов. **DispatchMessage** направляет сообщение соответствующей оконной процедуре.

Оконная процедура

Мы уже упоминали ранее, что оконная процедура — это то место, где мы указываем код, который должен выполняться при получении окном приложения каких-либо сообщений. В программе Hello World оконная процедура называется **WndProc**. Ее прототип выглядит так:

```
LRESULT CALLBACK WndProc(
     HWND hwnd,
     UINT uMsg,
     WPARAM wParam,
     LPARAM lParam
);
```

Функция возвращает значение типа **LRESULT** (в действительности это целое число типа **long**), сообщающее успешно или нет завершена работа функции. Идентификатор **CALLBACK** сообщает, что это *функция обратного вызова* (*callback function*). Это означает, что вызов данной функции осуществляют внутренние механизмы Windows. Посмотрите на исходный код приложения Hello World: вы нигде не найдете явно указанного нами вызова оконной процедуры; Windows вызывает ее за нас, когда окну требуется обработать поступившее сообщение.

В сигнатуре оконной процедуры указано четыре параметра:

- **hwnd** — Идентифицирует окно, которому предназначено сообщение.
- **uMsg** — Предопределенная константа, идентифицирующая конкретное сообщение. Например, сообщению о выходе из приложения соответствует константа **WM_QUIT**. Префикс **WM** означает «оконное сообщение» (**W**indow **M**essage). Существуют сотни предопределенных оконных сообщений, описание которых можно найти в MSDN.
- **wParam** — Дополнительная информация о сообщении. Зависит от конкретного сообщения.
- **lParam** — Дополнительная информация о сообщении. Зависит от конкретного сообщения.

Наша оконная процедура обрабатывает три сообщения: **WM_LBUTTONDOWN**, **WM_KEYDOWN** и **WM_DESTROY**. Сообщение **WM_LBUTTONDOWN** посылается когда пользователь нажимает левую кнопку мыши и при этом указатель мыши находится внутри клиентской области окна. Сообщение **WM_KEYDOWN** отправляется если нажата какая-нибудь клавиша на клавиатуре. Сообщение **WM_DESTROY** будет получено в том случае, если окно должно быть уничтожено. Наш код очень простой; получив сообщение **WM_LBUTTONDOWN** мы выводим окно сообщений с текстом «Hello, World»:

```
case WM_LBUTTONDOWN:
     ::MessageBox(0, "Hello, World", "Hello", MB_OK);
     return 0;
```

Когда окно получает сообщение **WM_KEYDOWN** мы проверяем какая именно клавиша была нажата. Параметр **wParam**, передаваемый в оконную процедуру, содержит *виртуальный код клавиши* (*virtual key kode*), указывающий какая клавиша нажата. Вы можете думать о виртуальном коде клавиши как об идентификаторе конкретной клавиши на клавиатуре. В заголовочном файле Windows содержится список виртуальных кодов клавиш, которые мы можем использовать при проверке того, какая именно клавиша была нажата (например, чтобы проверить была ли нажата клавиша **Esc**, мы используем константу виртуального кода клавиши **VK_ESCAPE**).

ПРИМЕЧАНИЕ Помните, что параметры **wParam** и **lParam** используются для передачи дополнительной информации, относящейся к конкретному сообщению. Для сообщения **WM_KEYDOWN** переменная **wParam** содержит виртуальный код нажатой клавиши. В библиотеке MSDN описано какую именно информацию содержат переменные **wParam** и **lParam** для каждого сообщения Windows.

```
case WM_KEYDOWN:
     if( wParam == VK_ESCAPE )
          ::DestroyWindow(MainWindowHandle);
     return 0;
```

Когда наше окно должно быть уничтожено, мы отправляем сообщение о выходе из приложения (которое прерывает работу цикла сообщений).

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

В самом конце оконной процедуры мы вызываем функцию **DefWindowProc**. Это стандартная оконная процедура. В приложении Hello World мы обрабатываем только три сообщения; стандартная оконная процедура определяет поведение при получении всех остальных сообщений, которые мы получаем, но решили не обрабатывать самостоятельно. Например, окно приложения Hello World может быть свернуто, развернуто на весь экран, закрыто, может быть изменен его размер. Вся эта функциональность предоставляется нам стандартной оконной процедурой, и нам не надо писать свои обработчики сообщений, реализующие эти функции. Обратите внимание, что функция **DefWindowProc** является частью Win32 API.

Функция MessageBox

Последняя функция API, которую мы сейчас рассмотрим — это функция **MessageBox**. Она очень полезна в тех случаях, когда надо показать какую-то информацию пользователю и получить от него ответ. Прототип функции **MessageBox** выглядит так:

```
int MessageBox(
     HWND    hWnd,       // Дескриптор владельца окна,
                         // можно указать null
     LPCTSTR lpText,     // Текст, выводимый в окне
     LPCTSTR lpCaption, // Текст, выводимый в заголовке окна
     UINT    uType       // Стиль окна сообщения.
);
```

Значение, возвращаемое функцией **MessageBox** зависит от типа окна сообщения. Стили окон сообщений и соответствующие им возвращаемые значения перечислены в MSDN.

Улучшенный цикл сообщений

Игры отличаются от обычных Windows-приложений, таких как офисные программы или браузеры. Игры не управляются событиями в традиционном понимании этого термина и должны постоянно выполнять обновление. Поэтому, когда мы начнем писать наши приложения с трехмерной графикой, работа с сообщениями Windows не будет центром разработки. Следовательно, нам надо модифицировать цикл обработки сообщений таким образом, чтобы когда придет сообщение мы смогли обработать его. Но в то же время, если сообщений нет, должен выполняться код нашей игры. Вот как выглядит новый цикл сообщений:

```
int Run()
{
     MSG msg;

     while(true)
     {
          if(::PeekMessage(&msg, 0, 0, 0, PM_REMOVE))
          {
               if(msg.message == WM_QUIT)
                    break;

               ::TranslateMessage(&msg);
               ::DispatchMessage(&msg);
          }
          else
               // выполнение кода игры
     }
     return msg.wParam;
}
```

После объявления переменной **msg**, мы попадаем в бесконечный цикл. Сперва мы вызываем функцию API **PeekMessage**, которая проверяет наличие сообщений в очереди. Описание аргументов этой функции смотрите в MSDN. Если в очереди есть сообщение, функция возвращает **true**, после чего мы обрабатываем полученное сообщение. Если **PeekMessage** возвращает **false**, мы выполняем код нашей игры.

Итоги

- Чтобы использовать Direct3D, мы должны создать Windows-приложение с главным окном, в котором будет осуществляться визуализация наших трехмерных сцен. Кроме того, для игр следует написать специализированный цикл обработки сообщений, который будет проверять наличие в очереди сообщений и, если они там есть, обрабатывать их; когда сообщений в очереди нет должен выполняться относящийся к игре код.
- В Windows могут одновременно работать несколько приложений; поэтому Windows должна управлять распределением ресурсов между ними и направлять сообщения тому приложению, которому они предназначены. Сообщения помещаются в очередь сообщений приложения когда происходит какое-либо событие (нажатие клавиши, щелчок кнопки мыши, срабатывание таймера и т.д.), относящееся к этому приложению.
- У каждого Windows-приложения есть собственная очередь сообщений, где хранятся полученные приложением сообщения. Цикл обработки сообщений постоянно проверяет наличие сообщений в очереди и отправляет их соответствующей оконной процедуре. Обратите внимание, что у одного приложения может быть несколько окон.
- Оконная процедура — это специальная реализуемая разработчиком приложения функция обратного вызова, к которой обращается Windows, когда окно приложения получает сообщение. В оконной процедуре мы пишем код, который должен быть выполнен в том случае, если окно приложения получило данное сообщение. Сообщения, для которых мы не задали действия по их обработке, перенаправляются стандартной оконной процедуре, которая выполняет заданную по умолчанию обработку.

Список литературы

- Edward Angel «Interactive Computer Graphics: A Top-Down Approach with OpenGL» 2-е изд. Addison-Wesley, 2000

  Есть русский перевод: Эдвард Энджел «Интерактивная компьютерная графика. Вводный курс на базе OpenGL» 2-е изд. Вильямс, 2001
- Jim Blinn «Jim Blinn's Corner: A Trip Down the Graphics Pipeline», стр. 53-61. San Francisco: Morgan Kaufmann Publishers, Inc., 1996
- David H. Eberly «3D Game Engine Design» San Francisco: Morgan Kaufmann Publishers, Inc., 2001
- Wolfgang F. Engel, ред. «Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks» Plano, Texas: Wordware Publishing, 2002
- Fraleigh и Beauregard «Linear Algebra» 3-е изд. Addison-Wesley, 1995
- Mark J. Kilgard «Creating Reflections and Shadows Using Stencil Buffers» Game Developers Conference, презентация NVIDIA, 1999. (*http://developer.nvidia.com/docs/IO/1407/ATT/stencil.ppt*)
- Jeff Lander «Shades of Disney: Opaquing a 3D World» Game Developer Magazine, March 2000
- Eric Lengyel «Mathematics for 3D Game Programming & Computer Graphics» Hingham, Mass.: Charles River Media, Inc., 2002
- Microsoft Corporation. Документация SDK Microsoft DirectX 9.0. Microsoft Corporation, 2002
- Tomas Moller, Eric Haines «Real-TimeRendering» 2-е изд. Natick, Mass.: A K Peters, Ltd., 2002
- M.E. Mortenson «Mathematics for Computer Graphics Applications» 2-е изд. New York: Industrial Press, Inc., 1999
- Charles Petzold «Programming Windows» 5-е изд. Redmond, Wash.: Microsoft Press, 1999
- Jeff Prosise «Programming Windows with MFC» 2-е изд. Redmond, Wash.: Microsoft Press, 1999
- Sergei Savchenko «3D Graphics Programming: Games and Beyond» стр. 153-156, 253-294. Sams Publishing, 2000
- John van der Burg «Building an Advanced Particle System» Gamasutra, June 2000. (*http://www.gamasutra.com/features/20000623/vanderburg_01.htm*)
- Alan Watt, Fabio Policarpo «3D Games: Real-time Rendering and Software Technology» Addison-Wesley, 2001

- Alan Watt «3D Computer Graphics» 3-е изд. Addison-Wesley, 2000
- Gabriel Weinreich «Geometrical Vectors» стр. 1-11. Chicago: The University of Chicago Press, 1998
- Nicholas Wilt «Object-Oriented Ray Tracing in C++» стр. 56-58. New York: John Wiley & Sons, Inc., 1994

Алфавитный указатель

3

A

B

C

D