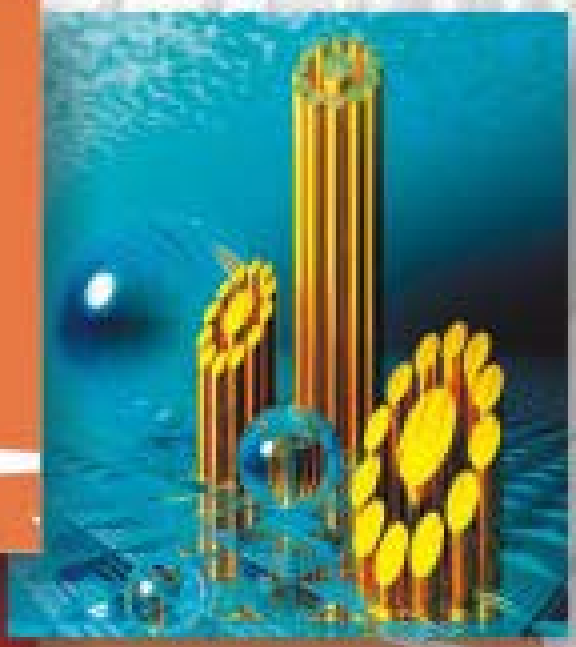
СТАНИСЛАВ ГОРНАКОВ

bhv

ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА ПРОГРАММИРОВАНИЯ И ОТЛАДКИ ШЕЙДЕРОВ В DirectX И OpenGL



ОСНОВЫ DirectX

АССЕМБЛЕРНЫЙ ЯЗЫК ШЕЙДЕРОВ

ПРОФАЙЛЕР PIX for Windows

ATI RenderMonkey 1.5

NVIDIA FX Composer 1.5

PRO

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

+CD

УДК 681.3.06
ББК 32.973.26-018.2
Г67

Горнаков С. Г.

Г67 Инструментальные средства программирования и отладки шейдеров в DirectX и OpenGL. — СПб.: БХВ-Петербург, 2005. — 256 с.: ил.

ISBN 5-94157-611-0

Рассматриваются основы DirectX, показаны приемы работы с фиксированным и программируемым графическими конвейерами, дана информация по применению профайлера PIX for Windows, необходимого для отладки программ в DirectX, подробно представлена информация об инструментариях ATI RenderMonkey 1.5 и NVIDIA FX Composer 1.5, позволяющих создавать и отлаживать шейдеры в DirectX и OpenGL.

Для программистов графики

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Рыбинский*
Зав. редакцией	*Григорий Добин*
Редактор	*Екатерина Капалыгина*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 26.05.05.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 20,64.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

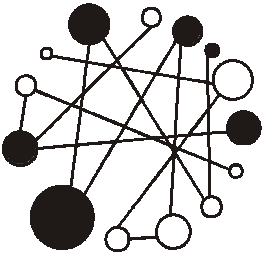
Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

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

ISBN 5-94157-611-0

Оглавление

ГЛАВА 2



Фиксированный графический конвейер

Как уже упоминалось в предыдущей главе, для построения сцены в DirectX 9 необходимо выполнить последовательно шаг за шагом определенные действия. Набор этих операций можно разбить на несколько следующих этапов:

1. *Создание оконного приложения.* На этом этапе необходимо создать простое оконное приложение, в которое впоследствии можно встраивать исходный код создаваемой сцены. Оконное приложение создается на основе библиотечных функций Win32 API.
2. *Инициализация и настройка Direct3D9.* На этом этапе происходит инициализация Direct3D9, в том числе создание необходимых указателей на интерфейсы и создание устройства Direct3D9. Также необходимо произвести настройку частоты смены кадров и разрешение буфера глубины, т. е. задать необходимые параметры для представления графики на экране монитора.
3. *Создание сцены.* После инициализации и настройки Direct3D9 можно приступить к созданию или загрузке объектов и всей сцены в частности. При создании объектов используются вершинные и индексные буферы, где хранятся позиции точек, из которых состоит объект. Для загрузки готового объекта, созданного с помощью любого редактора трехмерной графики, в DirectX используются X-файлы. Конвертировав готовую модель в X-формат, вы сможете легко загрузить ее с помощью функций DirectX 9. Также есть возможность загрузки объектов и других форматов.
4. *Трансформация и освещение (T&L).* Этот этап, пожалуй, самый сложный и именно на этой стадии вы можете задействовать фиксированный либо программируемый графический конвейер. Для представления объекта на экране применяются матричные преобразования, а для того чтобы объект выглядел реалистично, используются материал и освещение.

5. *Рендеринг сцены*. После инициализации, настройки Direct3D и построения сцены можно производить вывод графики на экран монитора.

Все рассмотренные этапы создания и отрисовки трехмерной графики не включали в себя использование клавиатуры, мыши и звука. Для того чтобы задействовать перечисленные компоненты, необходимо выполнить еще ряд дополнительных действий. Эти процессы в книге не рассматриваются, но при необходимости вы сможете найти нужную информацию в книге "DirectX 9. Уроки программирования на C++" издательства "БХВ-Петербург".

Двойная буферизация

Смысл *двойной буферизации* чрезвычайно прост. Для вывода графики на экран используется *задний буфер* (back buffer), в который помещается очередной кадр. Задний буфер по своим параметрам, разрешению и цветности идентичен первичной поверхности экрана, и за счет постоянной смены буферов или переключения поверхностей достигается плавная анимация в графике. Например, вы вывели на экран некую модель и желаете переместить ее в другой конец экрана. Для этого необходимо стереть модель в месте ее первоначального вывода и нарисовать заново. Если производить эти действия на первичной поверхности, анимация может происходить с заметными глазу рывками. Куда проще нарисовать модель на новом месте в заднем буфере и произвести смену буферов. Этот процесс вначале может показаться сложным, но на самом деле программисту не приходится отслеживать задний буфер, эти действия выполняются программно средствами DirectX. Просто в определенных параметрах представления устройства выставляются необходимые флаги и количество создаваемых задних буферов, после чего процесс двойной буферизации будет осуществляться автоматически под контролем DirectX.

Теперь давайте рассмотрим подробнее стадии по созданию и отрисовке сцены на экране монитора.

Создание оконного приложения

Операционная система Windows работает на основе управляемых событий, поэтому, создавая *оконное приложение*, вы также должны позаботиться о создании обработчика событий. Создавая оконное приложение, вы вправе определить стиль, размер, цвет фона и режимы отображения оконного приложения. Разбор исходного кода, иллюстрирующего создание оконного приложения, к сожалению, выходит за рамки этой книги, но в конце данной главы и на компакт-диске к книге в папке \Code\Demo в файле Demo.cpp вы можете ознакомиться с кодом примера, создающим оконное

приложение, а также полным спектром действий по инициализации, настройке, созданию объекта и рендерингу с использованием Direct3D9. Исходный код несложен, и я думаю, вам не составит труда разобраться во всем самостоятельно.

Инициализация и настройка Direct3D9

Процесс *инициализации* и *настройки Direct3D9* сводится к последовательному вызову библиотечных функций и заполнению полей структуры параметров представления Direct3D9. Но первым делом необходимо создать *указатель* на главный интерфейс `IDirect3D9`.

```
LPDIRECT3D9 pd3d9 = NULL;
pd3d9 = Direct3DCreate9( D3D_SDK_VERSION);
```

После объявления переменной `pd3d9` (предварительно обнулив ее значение) нужно воспользоваться функцией `Direct3DCreate()` для создания указателя на интерфейс `IDirect3D9`, который будет находиться в переменной `pd3d9`. Функция `Direct3DCreate()` в качестве параметра всегда использует макрос `D3D_SDK_VERSION`, указывающий на текущую версию DirectX SDK. Затем необходимо определить формат поверхности дисплея или текущий режим визуального отображения дисплея. Для этого нужно воспользоваться функцией `IDirect3D9::GetAdapterDisplayMode` и структурой `DISPLAYMODE`.

```
D3DDISPLAYMODE Display;
pd3d9->GetAdapterDisplayMode(D3DADAPTER_DEFAULT, &Display);
```

Функция `GetAdapterDisplayMode()` возвращает установленный текущий формат поверхности дисплея, сохраняя полученный результат в переменной `Display`, а именно: разрешение экрана (т. е. ширину и высоту), цветность и частоту обновления экрана. Все полученные данные необходимы для создания заднего буфера, который должен быть идентичен исходной поверхности дисплея.

> ***Примечание***
>
> При работе с библиотечными функциями DirectX 9 необходимо использовать отлаженную систему обработки ошибок. Для этого в DirectX предусмотрены два макроса: `FAILED` — проверяющий программный код на наличие ошибок и `SUCCEEDED` — проверяющий программный код на успешность выполнения. Оба макроса имеют соответствующие коды возврата ошибок, о которых можно узнать из справочной информации DirectX SDK. Например, для того чтобы определить текущий формат дисплея, можно воспользоваться следующей конструкцией кода:
>
> ```
> if(FAILED(pd3d9->GetAdapterDisplayMode(D3DADAPTER_DEFAULT,
>    ↳dis)))return E_FAIL;
> ```

Следующим шагом в процессе настройки Direct3D9 будет установка параметров представления Direct3D9 с помощью структуры `D3DPRESENT_PARAMETERS`. Это очень большая структура, с ее помощью создаются задний буфер, буфер глубины и определяется полноэкранный или оконный режим работы приложения.

```
D3DPRESENT_PARAMETERS d3d9pp;
// обнуление
ZeroMemory(&d3d9pp, sizeof(d3d9pp));
// задействуется полноэкранный режим
d3d9pp.Windowed = FALSE;
// подключается задний буфер
d3d9pp.SwapEffect = D3DSWAPEFFECT_DISCARD;
// формат поверхности заднего буфера
d3d9pp.BackBufferFormat = Display.Format;
// создается буфер глубины
d3d9pp.EnableAutoDepthStencil = TRUE;
// формат поверхности буфера глубины
d3d9pp.AutoDepthStencilFormat = D3DFMT_D16;
// ширина заднего буфера
d3d9pp.BackBufferWidth  = Display.Width;
// высота заднего буфера
d3d9pp.BackBufferHeight = Display.Height;
// количество задних буферов
d3d9pp.BackBufferCount = 2;
// частота обновления
d3d9pp.FullScreen_RefreshRateInHz = Display.RefreshRate;
```

Приведенный исходный код имеет комментарии, и в нем будет нетрудно разобраться. Но хочется добавить замечание по поводу заднего буфера и буфера глубины. Как вы уже знаете, задний буфер необходим для создания качественной анимации в компьютерных играх. В параметре

```
d3d9pp.BackBufferCount = 2;
```

было создано два задних буфера, но их число может быть и больше, главное чтобы ваш видеоадаптер поддерживал указанное количество задних буферов. Два-три задних буфера — это вполне стандартное значение.

А для чего же нужен *буфер глубины*? Экран монитора по своей сути представляет собой абсолютно плоскую двухмерную поверхность, и для того чтобы правильно отображать трехмерные модели, применяется буфер глубины, который иногда еще называют *z-буфер*. То есть это дополнительная ось Z, необходимая для правильного представления объемной трехмерной графики на экране монитора.

После создания z-буфера нужно созданный буфер подключить в приложение, а также задействовать режим отсечения невидимых граней объекта.

```
pDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);
pDevice->SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
```

Подключение опции *отсечения невидимых граней* объекта — это общепринятая практика в DirectX. Например, вы рисуете на экране куб, вращающийся вокруг своей оси. В какой-то промежуток времени одна из частей куба обязательно закроет какую-то другую часть куба. И поскольку вы все равно не увидите эту закрытую часть, то включается режим отсечения невидимых граней. Зачем же тратить ресурсы процессора и памяти на те части объекта, которые не видно. Сам процесс по отсечению происходит автоматически, главное только правильно построить сам объект.

Последний шаг в настройке и инициализации Direct3D9 заключается в создании устройства Direct3D9 с помощью функции `IDirect3DDevice9::CreateDevice`.

```
pd3d9 -> CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hwnd,
D3DCREATE_HARDWARE_VERTEXPROCESSING, &d3d9pp, &pDevice);
```

Инициализировав и настроив Direct3D9, можно переходить к этапу по созданию сцены.

Создание сцены

Построение *сцены* в Direct3D — это процедура трудоемкая и всецело зависит от задачи, которую вам необходимо решить. На этом этапе возможна загрузка готовых трехмерных моделей, созданных, например, в 3D Studio Max, или целого уровня, созданного в том же редакторе, или построение объекта сервисами Direct3D, которые мы и рассмотрим.

В качестве примера возьмем программу Demo, код которой находится в конце главы. В этом примере происходит построение куба на основе индексного и вершинного буфера. Концепция построения объектов в DirectX использует понятие *вершина*. Каждый объект, каким бы он не был сложным, состоит из полигонов. *Полигон* — это определенного размера площадь в пространстве, ограниченная точками. В Direct3D обычно используется треугольник, площадь которого ограничивается тремя углами. Каждый угол имеет свои координаты в пространстве по осям X, Y и Z. Точка, заданная в пространстве по трем осям X, Y и Z, в DirectX называется *вершиной*. Определив значения для трех вершин, можно построить простой треугольник, а соединив два треугольника по гипотенузе, получаем квадрат. Чем больше полигонов, тем сложнее фигура для построения модели.

В DirectX могут применяться *системы координат* двух видов, в зависимости от установленного формата вершин. Определение формата вершин зависит

от того, будет ли рисоваться двухмерная или трехмерная графика. При использовании двухмерных объектов применяется система координат, изображенная на рис. 2.1, и в этом случае при построении объекта используется *преобразованный формат вершин*.

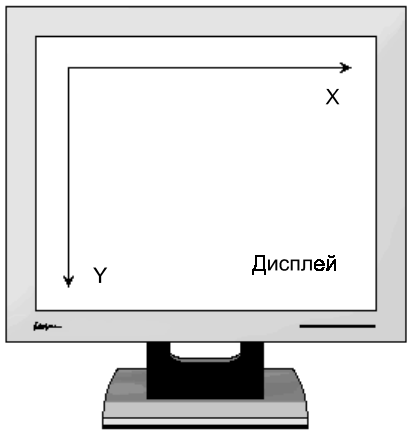


Рис. 2.1. Двухмерная система координат

При создании трехмерного объекта применяется более сложная модель работы с вершинами. Вершины оставляют *не преобразованными*, а для визуализации вершин на экране монитора используются матричные преобразования. Поэтому применяется левосторонняя система координат, показанная на рис. 2.2.

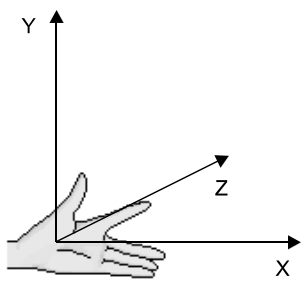


Рис. 2.2. Левосторонняя система координат

Но обо всем по порядку. Создаваемый объект состоит из набора полигонов, задающихся вершинами. Позиция одной вершины задается тремя координатами по осям X, Y и Z. Для построения квадрата необходимо два соприкасающихся гипотенузами треугольника. Для создания двух треугольников потребуется шесть вершин, а для того чтобы построить куб, понадобится

6 вершин * 6 сторон куба = 36 вершин. Для хранения такого количества вершин логично использовать структуру, например:

```
struct CUSTOMVERTEX
{
    FLOAT x,  y,  z;
    FLOAT nx, ny, nz;
};
```

Координаты `x`, `y` и `z` определяют позицию вершины в пространстве, а три других координаты определяют направление нормалей, необходимых для правильного расчета освещения сцены, о котором вы узнаете в этой главе из *разд. "Трансформация и освещение"*.

Одна сторона куба — это два соприкасающихся треугольника. Для построения одной стороны, т. е. квадрата, необходимо задать позиции для шести вершин, по три на каждый треугольник. При детальном рассмотрении выясняется, что две вершины каждого треугольника будут иметь одинаковые позиции. Поэтому можно значительно упростить процесс построения квадрата, и куба в частности, путем *индексации* каждой вершины, а полученные значения хранить в отдельном массиве данных. В конце главы и на компакт-диске в папке \Code\Demo\Demo.cpp вы можете ознакомиться с кодом, создающим индексированный куб с помощью структуры `Vertex` и массива индексов `Index`.

После того, как были заданы позиции для всех вершин, необходимо указать используемый формат вершин. Для этого применяется следующая конструкция программного кода:

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

Макрос `D3DFVF_XYZ` указывает на не преобразованный формат вершин, поскольку все преобразования будут осуществляться с помощью матриц. Если вы желаете использовать формат преобразованных вершин, то нужно воспользоваться макросом `D3DFVF_XYZRHW`.

Последующие действия по построению куба сводятся к использованию индексного буфера и буфера вершин. Буфер вершин необходим для хранения общего количества вершин объекта, можно сказать, что это заданный массив данных, в котором содержатся сами вершины, а также цвет каждой вершины и нормали. Впоследствии при рендеринге буфер вершин используется для вывода объекта на экран. Индексный буфер содержит индексы для каждой стороны куба и также используется при рендеринге. Процесс работы с буфером вершин и индексным буфером сводится к нескольким последовательным шагам — это создание буфера, его блокировка для последующей записи данных и разблокировка буфера. В следующих строках кода показаны перечисленные операции для буфера вершин:

```
LPDIRECT3DVERTEXBUFFER9 pBV = NULL;
// создание буфера
pDevice->CreateVertexBuffer(36 * sizeof(CUSTOMVERTEX),
        0, D3DFVF_CUSTOMVERTEX,D3DPOOL_DEFAULT, &pBV, NULL);
VOID* copy;
// блокировка буфера
pBV->Lock(0, sizeof(Vertex), (void**)&copy, 0);
// запись данных в буфер
memcpy(copy, Vertex, sizeof(Vertex));
// разблокировка буфера
pBV->Unlock();
```

Для записи данных в буфер вершин необходимо обязательно заблокировать его, а каждая блокировка буфера должна сопровождаться его разблокировкой. Действия по работе с индексным буфером практически идентичны действиям с буфером вершин.

```
LPDIRECT3DINDEXBUFFER9 pBI = NULL;
// создание буфера
pDevice->CreateIndexBuffer(36 * sizeof(Index),
      0, D3DFMT_INDEX16, D3DPOOL_DEFAULT,&pBI, NULL);
// блокировка буфера
pBI->Lock(0, sizeof(Index), (void**)&copy, 0);
// запись данных в буфер
memcpy(copy, Index, sizeof(Index));
// разблокировка буфера
pBI->Unlock();
```

Таким образом, после создания объекта можно перейти к следующему этапу.

Трансформация и освещение

На этом этапе можно выбрать работу с фиксированным или программируемым графическим конвейером. Программируемый конвейер использует вершинные шейдеры и будет обсуждаться в следующей главе. Основная база, на которой основана работа с фиксируемым и программируемым графическими конвейерами, построена на использовании матричных преобразований и работе с материалом и освещением.

Матричные преобразования

Для представления трехмерной сцены на экране монитора в DirectX задействуются матричные преобразования. Матричные операции основаны на использовании *матриц* — это мировая матрица (World Matrix), матрица вида

(View Matrix) и матрица проекции (Projection Matrix). Все три матрицы имеют вид массивов данных с размерностью 4×4.

Для того чтобы правильно отобразить объект на экране, необходимо преобразовать каждую вершину с помощью последовательного использования трех матриц в следующем порядке: мировая матрица, матрица вида и матрица проекции. Создание каждой из матриц можно выполнить вручную, а можно воспользоваться достаточно большим набором функций из библиотеки утилит D3DX, входящей в состав Direct3D9.

> ***Примечание***
>
> Для того чтобы воспользоваться библиотекой D3DX, необходимо подключить заголовочный файл d3dx9.h и библиотечный файл d3dx9.lib в проект.

Мировая матрица

Мировая матрица — это первая матрица, на которую нужно умножить каждую вершину объекта. С помощью мировой матрицы производится мировое преобразование объекта, и в результате каждый из объектов получает свою локальную систему координат. На основании локальной системы координат происходит построение объекта. Также при использовании мирового преобразования доступны действия по вращению, трансляции и масштабированию объекта. Например, для того чтобы преобразовать объект в мировое пространство и осуществить вращение по осям X и Y, можно воспользоваться следующим программным кодом.

```
D3DXMATRIX MatrixWorld, MatrixWorldX, MatrixWorldY;
FLOAT Angel = (timeGetTime() % 3000) * (2.0f * D3DX_PI) / 3000.0f;
D3DXMatrixRotationX(&MatrixWorldX, Angel);
D3DXMatrixRotationY(&MatrixWorldY, Angel);
D3DXMatrixMultiply(&MatrixWorld,&MatrixWorldX,&MatrixWorldY);
```

С помощью функций `D3DXMatrixRotationX()` и `D3DXMatrixRotationY()` производится вращение объекта по осям X и Y на угол, равный значению в переменной `Angel`. Полученный результат вращения по оси X сохраняется в переменной `MatrixWorldX`, а по оси Y — в переменной `MatrixWorldY`. После этого два результата вращения по разным осям перемножаются между собой с помощью функции `D3DXMatrixMultiply()`, а итоговый результат помещается в переменную `MatrixWorld`. Для того чтобы мировое преобразование вступило в силу, необходимо воспользоваться функцией `IDirect3DDevice9::SetTransform`.

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

Матрица вида

С помощью *матрицы вида* происходит определение позиции просмотра сцены. Например, за позицию просмотра сцены можно взять местонахож-

дение ваших глаз в пространстве, а поскольку ваш взгляд может быть устремлен в любую точку экрана монитора, то должно определяться направление взгляда и середина области просмотра, т. е. то, на что вы смотрите в данный момент. Для преобразования объекта матрицей вида нужно воспользоваться функцией `D3DXMatrixLookAtLH()` из библиотеки утилит D3DX.

```
D3DXMATRIX MatrixView;
D3DXMatrixLookAtLH(&MatrixView, &D3DXVECTOR3 (0.0f,0.0f,-11.0f),
&D3DXVECTOR3 (0.0f, 0.0f, 0.0f), &D3DXVECTOR3 (0.0f, 1.0f, 0.0f));
```

Первый параметр `MatrixView` в функции `D3DXMatrixLookAtLH()` — это результат видовых преобразований. Три последующих параметра определяют соответственно точку просмотра сцены, на что конкретно направлен взгляд и что в итоге мы видим. Для определения позиции просмотра используется структура `D3DXVECTOR3`, представляющая точку в пространстве по осям X, Y и Z.

Для того чтобы видовые преобразования вступили в силу, нужно воспользоваться функцией `SetTransform()`.

```
pDevice->SetTransform(D3DTS_VIEW, &MatrixView);
```

Матрица проекции

После преобразования объекта мировой матрицей и матрицей вида производится последний этап преобразований с помощью *матрицы проекции*. Проекционные преобразования позволяют правильно представить объект на двухмерной плоскости монитора, масштабируя объект относительно позиции просмотра, передней и задней области отсечения.

```
D3DXMATRIX MatrixProjection;
D3DXMatrixPerspectiveFovLH(&MatrixProjection,D3DX_PI/4,1.0f,1.0f,100.0f);
```

Первый параметр в функции `D3DXMatrixPerspectiveFovLH()` является результатом преобразований. Следующий параметр определяет поле зрения по оси Y, обычно это значение равно `D3DX_PI/4`. Третий параметр — коэффициент сжатия для соотношения геометрических размеров. Два последних параметра задают передний и задний планы отсечения сцены. Чтобы назначенные преобразования вступили в силу, вызывается функция `SetTransform()`.

```
pDevice->SetTransform(D3DTS_PROJECTION, &MatrixProjection);
```

После матричных преобразований можно воспользоваться освещением объекта для реалистичной отрисовки его на экране.

Освещение

Применение освещения в построении трехмерной сцены значительно улучшает качество рисуемой графики. В реальной жизни нас окружают множе-

ство разных предметов, каждый из которых имеет свой цвет, а точнее поверхность с определенным цветовым свойством. При попадании лучей света на поверхность предмета происходит гармоничное сочетание свойств света и свойств поверхности предмета, что в итоге и предопределяет наши цветовые и световые ощущения, полученные от этого предмета. В Direct3D поверхность объекта с цветовыми свойствами называется *материалом*. При работе с освещением необходимо в обязательном порядке задавать свойства материала для объекта, иначе построение сцены с использованием источников света не будет иметь никакого эффекта.

Материал

Материал в Direct3D9 создается на основе структуры `D3DMATERIAL9`, имеющей пять параметров, задавая значения которых можно определить необходимые свойства материала объекта.

Параметры структуры `D3DMATERIAL9` определяют:

- `Diffuse` — диффузный цвет материала;
- `Ambient` — окружающий цвет;
- `Specular` — отражающий цвет;
- `Emissive` — эмиссионный цвет;
- `Power` — мощность отражаемого цвета.

Четыре вида цвета `Diffuse`, `Ambient`, `Specular` и `Emissive` задаются с помощью структуры `D3DCOLORVALUE`, которая содержит ARGB составляющую цвета. Задавать значения красного, синего, зеленого цветов и альфа канала можно в пределах от 0.0 до 1.0. Более темные тона можно получить, если использовать значение меньше 0.0, и наоборот, светлые тона получаются при использовании значения больше 1.0. В следующих строках кода создается материал желтого цвета.

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

В последней строке кода используется функция `SetMaterial()` для установки назначенных свойств материала. После определения материала можно приступить к созданию источника света.

Свет

Как и в окружающем нас мире, в DirectX тоже предусмотрены источники света. Имеются три вида *источников света*:

- параллельный или направленный (directional) — этот тип не имеет определенного источника, но светит в определенном направлении;
- точечный (point-source) — имеет заданный источник света и светит во все направления;
- прожекторный (spotlight) — направленный источник света конусообразной формы, такой как, например, фонарик или прожектор.

Свет, излучаемый источником, имеет свои цветовые свойства, определяющие цвет излучаемого света. Свойства света задаются с помощью структуры `D3DLIGHT9`, например, следующим образом:

```
D3DLIGHT9 Light;
ZeroMemory(&Light, sizeof(D3DLIGHT9));
Light.Type = D3DLIGHT_DIRECTIONAL;
Light.Diffuse.r  = 1.0f;
Light.Diffuse.g  = 1.0f;
Light.Diffuse.b  = 1.0f;
Light.Range      = 1000.0f;
```

В первой строке программного кода задается значение для параметра `Type` структуры `D3DLIGHT9`. Этот параметр — член структуры `D3DLIGHTTYPE`, на основе которой происходит определение типа источника света. В конкретном случае используется макрос `D3DLIGHT_DIRECTIONAL`, назначающий параллельный или направленный источник света. Существуют еще два макроса для назначения источника света, это:

- `D3DLIGHT_POINT` — точечный источник света;
- `D3DLIGHT_SPOT` — прожекторный источник света.

В следующих трех строках исходного кода устанавливаются свойства цвета для источника излучения — это диффузный цвет. И в последней строке кода устанавливается дальность излучения источника света.

Для того чтобы свет начал работать в освещении сцены, необходимо назначить созданный источник света для устройства Direct3D, разрешить работу с освещением и установить дополнительные свойства света.

```
pDevice->SetLight(0, &Light);
pDevice->LightEnable(0, TRUE);
pDevice->SetRenderState(D3DRS_LIGHTING, TRUE);
pDevice->SetRenderState(D3DRS_AMBIENT, 0);
```

Нормализация

Последний этап в процессе освещения — это нормализация вершин рисуемого объекта. *Нормаль* — это вектор, расположенный перпендикулярно стороне или вершине примитива. Нормаль принимает непосредственное участие в расчете освещения сцены. Каждая вершина объекта должна быть нормализована по отношению к стороне примитива. Например, передняя часть куба в нашем примере имеет четыре вершины. Все вершины нормализованы, т. е. каждой вершине назначен вектор или нормаль, расположенная перпендикулярно стороне куба. Все последующие стороны куба должны быть нормализованы таким же образом. Для этого в структуре `CUSTOMVERTEX` и задаются нормали.

```
struct CUSTOMVERTEX
{
    FLOAT x,  y,  z;
    FLOAT nx, ny, nz; // нормали
};
```

Определяя формат вершин, необходимо задать и нормализацию.

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

Значение нормали в Direct3D определяется единичным значением, поэтому в структуре `Vertex` назначается единица для всех нормалей вершин. В конце этой главы найдите функцию `TL()`, структуру `Vertex` и посмотрите, каким образом происходит нормализация сторон куба в проекте Demo.

Для того чтобы нормали участвовали в расчете освещения, необходимо их установить для используемого источника света.

```
D3DXVECTOR3 VectorDir;
VectorDir = D3DXVECTOR3(0.0f, 0.0f,1.0f),
D3DXVec3Normalize((D3DXVECTOR3*)&Light.Direction, &VectorDir);
```

На этом этап по трансформации и освещению закончен, можно переходить непосредственно к выводу объекта на экран монитора.

Рендеринг сцены

Операции по рендерингу сцены лучше разместить в одной функции. Перед тем как производить прорисовку сцены, необходимо очистить задний буфер цветовой составляющей, а поскольку функция рендеринга впоследствии будет помещена в цикл Windows, то очистка заднего буфера цветом будет осуществляться циклично. Для очистки заднего буфера цветом воспользуемся функцией `IDirect3DDevice9::Clear`.

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

После очистки заднего буфера можно приступать к прорисовке всей сцены. Для этого Direct3D дается соответствующая команда:

```
pDevice->BeginScene();
```

Это, как вы догадались по названию функции `BeginScene()`, начало сцены для визуализации объекта. Далее связываем поток данных с буфером вершин:

```
pDevice->SetStreamSource(0, pBV, 0, sizeof(CUSTOMVERTEX));
```

Задаем формат вершин:

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

Производим установку индексов из буфера индексов для каждой вершины:

```
pDevice->SetIndices(pBI);
```

Затем осуществляем сборку куба:

```
pDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0, 36, 0, 12);
```

И даем команду на окончание сцены:

```
pDevice->EndScene();
```

На каждый вызов функции начала сцены `BeginScene()` должен следовать вызов функции окончания сцены `EndScene()`. Между двумя этими функциями происходит построение сцены в Direct3D. Для представления сцены на экране используется функция `IDirect3DDevice9::Present`.

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

И в конце, необходимо поместить функции по инициализации Direct3D, созданию объекта и рендеринга в цикл Windows.

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