МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
Факультет математики и информационных технологий

Д. А. Мальцев

Мультимедиатехнологии. Создание DirectX-приложений

Учебно-методическое пособие

Ульяновск
2013

УДК 004.9(075.8)
ББК 32.973.235я73+32.973.2-018.2я73
М21

Печатается по решению Ученого совета
факультета математики и информационных технологий
Ульяновского государственного университета

Рецензенты:
д. т. н., профессор ***К. В. Кумунжиев***;
к. ф.-м. н., доцент ***О. А. Фатьянова***

Мальцев, Д. А.
М21 Мультимедиатехнологии. Создание DirectX-приложений : учебно-методическое пособие / Д. А. Мальцев. – Ульяновск : УлГУ, 2013. – 44 с.

В учебно-методическом пособии рассматриваются основы разработки 3D-приложений на базе DirectX 10. Основной упор сделан на инициализацию контекста рендеринга, правильную загрузку ресурсов и рассмотрение структуры рендера.

Предназначено для студентов 3 курса факультета математики и информационных технологий специальностей «Прикладная математика» и «Информационные системы». Может быть использовано для самостоятельного изучения курса.

УДК 004.9(075.8)
ББК 32.973.235я73+32.973.2-018.2я73

Директор Издательского центра *Т. В. Филиппова*
Редактирование и подготовка оригинал-макета *Г. И. Петровой*

Подписано в печать .03.13. Формат 60×84/16. Усл. печ. л. ,. Уч.-изд. л. ,.
Тираж 100 экз. Заказ № /

Оригинал-макет подготовлен в Издательском центре
Ульяновского государственного университета
432017, г. Ульяновск, ул. Л. Толстого, 42

Отпечатано с оригинал-макета в Издательском центре
Ульяновского государственного университета
432017, г. Ульяновск, ул. Л. Толстого, 42

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Существует древняя китайская пословица: «Дорога в тысячу ли начинается с первого шага». В разработке любого 3D-приложения этим первым шагом будет инициализация контекста рендера. Данное учебно-методическое пособие позволит начинающим программистам правильно сделать первый и последующий шаги, не оступиться и не упасть в начале долгого и интересного пути.

Цель, поставленная перед учебно-методическим пособием, представляет собой доскональный разбор всех этапов создания простейшего приложения, использующего контекст Direct3D. Для учебно-методического пособия был написан специальный пример, части которого изучаются на протяжении каждой главы пособия. Его можно скачать по ссылке http://staff.ulsu.ru/maltsevda/fls/dx10tut1.zip.

В учебно-методическом пособии рассматривается приложение, написанное с использованием Microsoft Visual Studio 2010 и базирующееся на API DirectX 10. Однако аналогичная структура программы подойдет и для API DirectX 11, а возможно, и последующих версий.

Предполагается, что читатель знаком с методическими пособиями [1] и [2], а также обладает базовыми навыками программирования на ЯП C++ и разработки программ под ОС Microsoft Windows.

Для успешной работы с DirectX 10 необходимо установить на компьютер следующие программы и библиотеки:

1. Microsoft Visual Studio[1]. Чем новее версия, тем лучше. Пример, на котором основано данное учебно-методическое пособие, написан в версии MS VS 2010. В ней уже есть необходимые заголовочные файлы и статические библиотеки для Microsoft DirectX вплоть до 11-й версии. Но отсутствуют вспомогательные библиотеки D3DX, их нужно ставить отдельно.
2. Microsoft DirectX SDK[2] за июнь 2010 г. (June 2010). Этот SDK добавляет необходимые вспомогательные библиотеки D3DX10. Также вместе с SDK поставляется документация, примеры и много другой полезной информации.

---

[1] http://www.microsoft.com/visualstudio/eng/downloads
[2] http://www.microsoft.com/en-us/download/details.aspx?id=6812

3. Последняя версия драйверов для вашей видеокарты. Стандартная версия от Microsoft редко поддерживает все возможности современных видеокарт. Их можно найти на сайте производителя.

В учебно-методическом пособии пример рассматривается от общего к частному. То есть начинается все с точки входа в программу и заканчивается функцией рендеринга изображения. Следовательно, код примера нужно смотреть снизу вверх. Такой подход позволит заранее понять, какие ресурсы задействованы в процессе рендеринга и как они были получены.

ТОЧКА ВХОДА В ПРОГРАММУ

Прежде чем переходить непосредственно к написанию кода, нужно подключить необходимые заголовочные файлы и статические библиотеки. Нам понадобится следующее:

```
#include <windows.h>
#include <d3d10.h>
#include <d3dx10.h>
```

Файл *d3dx10.h* инсталлируется вместе с библиотекой DX SDK, остальные входят в стандартную установку Visual Studio. Статические библиотеки можно подключить в настройках проекта или таким образом:

```
#pragma comment(lib, "d3d10.lib")
#pragma comment(lib, "d3dx10.lib")
```

Любая программа имеет свою точку входа – функцию или процедуру, с которой начинается выполнение программы. Например, на языке C++ – это функция *main*, в паскале – блок кода между командами *begin* и *end*. Есть своя точка входа и у Windows-приложений, эта функция называется *WinMain*. Объявление этой функции выглядит следующим образом:

```
int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
      LPSTR lpCmdLine, int nCmdShow);
```

Чаще всего данная функция содержит три этапа:

1. Регистрация класса окна.
2. Создание окна.
3. Цикл обработки сообщений.

Подробнее о создании приложений для ОС Windows вы можете узнать из курса «Программирование в Windows». В данном учебно-методи-
ческом пособии мы лишь кратко пробежимся по каждому из этапов.

Регистрация класса окна

В Windows-приложениях нередко бывает ситуация, когда два различных окна выполняют одну и ту же функцию. Например, два разных окна Microsoft Word одинаковым способом позволяют редактировать текст; но при этом они содержат различные документы. Поэтому разработчики Windows разделили понятия *окно* и его *поведение*. За поведение окна отвечает его *класс*, а за отображение на экране необходимой информации и пользовательский ввод отвечает само *окно*. Основная задача при регистрации класса окна – указать процедуру обработки сообщений (*MainWndProc*), это происходит в 4-й строке листинга:

```
WNDCLASSEX wcex         = {0};
wcex.cbSize             = sizeof(WNDCLASSEX);
wcex.style              = CS_HREDRAW | CS_VREDRAW;
wcex.lpfnWndProc        = (WNDPROC)MainWndProc;
wcex.cbClsExtra         = 0;
wcex.cbWndExtra         = 0;
wcex.hInstance          = hInstance;
wcex.hIcon              = 0;
wcex.hCursor            = LoadCursor(NULL, IDC_ARROW);
wcex.hbrBackground      = (HBRUSH)(COLOR_WINDOW+1);
wcex.lpszMenuName       = NULL;
wcex.lpszClassName      = TEXT("dx10tutorial1");
wcex.hIconSm            = 0;
if ( RegisterClassEx(&wcex) == 0 )
      return 1;
```

Для идентификации класса окна ему назначается уникальное имя. Наш класс окна назван «dx10tutorial1». Имя нужно выбирать аккуратно, чтобы в пределах одной сессии Windows не было приложений с одинаковыми именами у различных классов окон.

За саму регистрацию отвечает функция *RegisterClassEx*. В случае успеха она возвращает ненулевой идентификатор класса окна, и его запоминать не обязательно. В случае ошибки – 0.

При регистрации класса окна можно указать курсор, цвет заливки и другие параметры.

Создание окна

Окно создается при помощи функции *CreateWindowEx*. Любое окно базируется на зарегистрированном классе окна (это второй параметр функции). Вдобавок при создании окну можно назначить размер, позицию, заголовок, стиль и т.п.

```
RECT rc = {0, 0, iWidth, iHeight};
AdjustWindowRect(&rc, WS_OVERLAPPED | WS_CAPTION |
      WS_SYSMENU | WS_MINIMIZEBOX, FALSE);

HWND hWnd = CreateWindowEx(0,
            TEXT("dx10tutorial1"),
            TEXT("DirectX 10 Tutorial 1"),
            WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX,
            CW_USEDEFAULT, CW_USEDEFAULT,
            rc.right - rc.left, rc.bottom - rc.top,
            NULL, NULL, hInstance, NULL);

if (hWnd == 0)
      return 2;

ShowWindow(hWnd, SW_SHOW);
UpdateWindow(hWnd);
```

В случае успеха функция *CreateWindowsEx* возвращает ненулевой идентификатор окна типа *HWND*. Его нужно запомнить, так как его требуют все функции для работы с окнами. Функции *ShowWindow* и *UpdateWindow* показывают окно на экране.

Осталось разобраться с самой первой функцией – *AdjustWindowRect*. Дело в том, что функция *CreateWindowEx* принимает размер окна, включающий в себя размер рамки, заголовка и т.п. То есть размер клиентской области окна, а значит, и размер DirectX-контекста, не будет совпадать с запрашиваемым – он будет меньше (см. рис. 1). Нужно понимать, что в различных версиях ОС Windows размер заголовка меняется, поэтому недостаточно просто прибавить какую-то константу к необходимому нам размеру клиентской области. *AdjustWindowRect* подбирает необходимый размер окна по его стилю и размеру клиентской области.

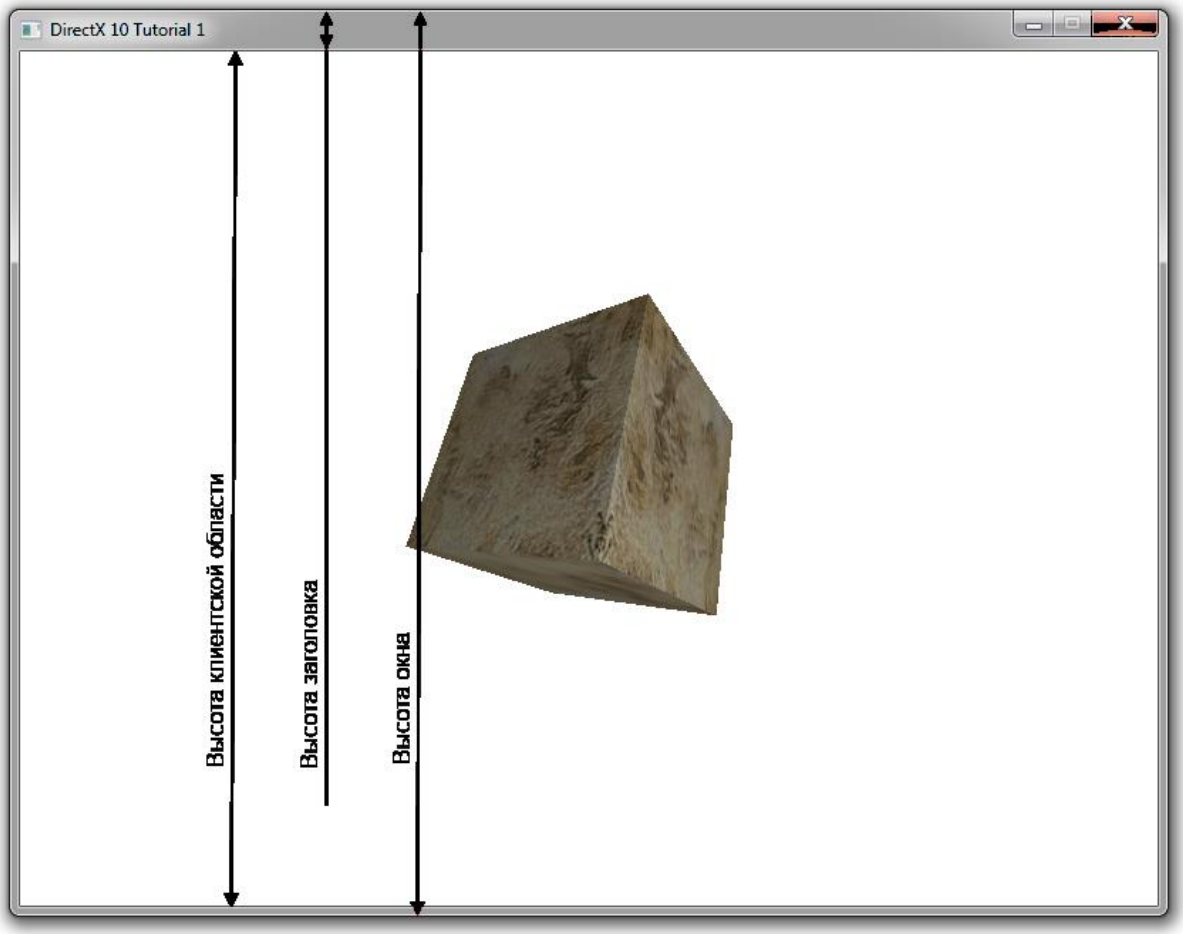

Рис. 1. Размер окна и размер клиентской области

Цикл обработки сообщений

Каждое окно непрерывно получает сообщения от системы, пользователя, других приложений и даже от самого себя. К таким сообщениям могут относиться сообщения о нажатии клавиш, сообщения о перемещении мыши, сообщения об изменении размеров окна, сообщения о создании и уничтожении окна и т.п. Все сообщения помещаются в очередь и постепенно вынимаются для последующей обработки. В случае если очередь пуста, то обычные приложения «замирают» – просто передают управление другим окнам в системе. Мультимедийные приложения в случае пустой очереди выполняют какие-то стандартные действия, например, вывод кадра на экран и расчет следующего кадра. Рассмотрим стандартный цикл обработки сообщений для мультимедийного приложения.

```
MSG msg = {0};
while (WM_QUIT != msg.message)
{
    while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE) == TRUE)
```

```
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

        if ( IsIconic(hWnd) == FALSE )
            Render();
    }
```

Структура *MSG* содержит всю необходимую информацию о сообщении. Параметр *message* этой структуры является уникальным идентификатором сообщения. Как видно из листинга выше, мы организуем бесконечный цикл до тех пор, пока этот параметр не станет равным значению *WM_QUIT* (это сообщение о том, что программа должна завершиться).

В теле цикла есть дополнительный цикл по обработке поступивших сообщений. *PeekMessage* извлекает сообщение из очереди. *TranslateMessage* преобразует виртуальный код клавиши в ее символьное значение. И *DispatchMessage* передает сообщение в процедуру обработки сообщений (см. следующую главу). Это стандартный способ обработки сообщений для Windows-приложений, и о нем подробно рассказано во всех учебных пособиях и статьях [5]. В случае если сообщений в очереди нет, *PeekMessage* возвращает *FALSE*.

Далее в теле цикла идет вызов функции *Render*. Это основная функция нашей программы, которая занимается генерацией 3D-картинки и выводом ее на экран, а также расчетом следующего кадра. Вызывается она только при условии, если функция *IsIconic* вернет *FALSE*. Это сделано для того, чтобы окно в минимизированном состоянии не занимало ресурсы процессора и видеокарты.

Подводя итог, суммируем все вышесказанное. Регистрируется класс окна, в котором указывается процедура обработки сообщений *MainWndProc*. На базе этого класса создается окно с определенным размером клиентской области. Организуется непрерывный цикл, в котором при наличии сообщений происходит их обработка, и вне зависимости от ситуации происходит вызов функции *Render*. Программа завершает свою работу при получении сообщения *WM_QUIT*.

ОКОННАЯ ПРОЦЕДУРА

Окно в ОС Windows непрерывно получает сообщения. Для их обработки программист должен реализовать специальную функцию, которая называется «процедурой обработки оконных сообщений» (window procedure). Объявление этой функции приведено ниже:

```
LRESULT CALLBACK MainWndProc(HWND hWnd, UINT uiMsg,
      WPARAM wParam, LPARAM lParam);
```

Чаще всего эта функция выглядит как организация выбора из множества различных вариантов. В данном мультимедийном приложении выбор будет простой:

- *WM_CREATE*, сообщение, которое посылается при создании окна;
- *WM_DESTROY*, сообщение, которое посылается при уничтожении окна;
- во всех остальных случаях будет вызываться функция *DefWindowProc*, реализующая стандартную обработку основных сообщений ОС Windows.

Рассмотрим листинг реализации этой функции:

```
case WM_CREATE:
      if ( InitDirectX10(hWnd) )
      {
            if ( InitResources() == FALSE )
                  return -1;
      }
      else
            return -1;
      break;

case WM_DESTROY:
      FreeResources();
      FreeDirectX10();
      PostQuitMessage(0);
      break;
```

При создании окна инициализируется контекст DirectX 10.0. В случае успешной инициализации происходит инициализация ресурсов приложения: текстур, шейдеров и т.п. Подробно обо всех этапах будет рассказано ниже. В случае ошибки оконная процедура вернет значение -1, что приведет к разрушению создаваемого окна и немедленному завершению программы.

При уничтожении окна освобождаются все ресурсы приложения и контекст Direct3D. Никаких исключительных ситуаций эти функции не предусматривают, т.е. разрушение ничем прервать нельзя. Финальная функция условия *PostQuitMessage* поместит в очередь сообщение с идентификатором *WM_QUIT*, что приведет к прерыванию цикла обработки сообщений и дальнейшему закрытию программы.

ОСНОВНЫЕ ПОНЯТИЯ COM-ТЕХНОЛОГИИ

API (Application Programming Interface – интерфейс прикладного программирования), работающий с DirectX, базируется на COM-технологии (от COM – англ. Component Object Model). Объем учебно-методического пособия не позволяет подробно описать эту технологию, поэтому книги, которые стоит изучить, указаны в библиографическом списке [6]. В этом разделе рассмотрим лишь основные понятия и принципы работы.

Основой COM-технологии являются интерфейсы. COM-интерфейс – это набор абстрактных функций, через который компоненты взаимодействуют друг с другом и с другими программами. Интерфейс задаётся абстрактным классом и реализуется другими, конкретными. Чаще всего интерфейс компонента самостоятельно создать нельзя, его можно или получить у специальной функции (такой как *CoCreateInstance*), или запросить у другого интерфейса. Из-за такой особенности работы удалить интерфейс тоже нельзя, его можно только освободить при помощи метода *Release*, который в зависимости от ситуации сам занимается удалением компонента и его интерфейсов.

Каждый интерфейс обладает своим уникальным идентификатором GUID. Он генерируется автоматически (с использованием специальных программ) и является уникальным во всем мире на все времена. Для того чтобы запросить интерфейс, необходимо передать этот идентификатор в метод или функцию.

Обработка ошибок базируется на возвращаемых значениях у методов. Это означает, что любой метод, который может сгенерировать исключительную ситуацию, возвращает код ошибки. В COM-технологии для этого задан специальный тип *HRESULT*, который может принимать различные значения как для успешной операции (например, *S_OK*), так и для исключительной (*E_NOINTERFACE*). Так как имеется несколько успеш-

ных кодов и немало исключительных, то для оценки результата выполнения функции применяют специальные макросы:

- *FAILED(result)* – возвращает истину, если результат содержит исключительную ситуацию;
- *SUCCEEDED(result)* – возвращает истину, если результат содержит успешное завершение работы метода.

В остальном работа с COM-технологией со стороны клиента не сильно отличается от обычного объектно-ориентированного программирования.

ИНИЦИАЛИЗАЦИЯ И ОСВОБОЖДЕНИЕ DirectX 10

В этой главе будут рассмотрены функции *InitDirectX 10* и *FreeDirectX 10*. Первая отвечает за получение интерфейса у Direct3D-устройства, создание интерфейса DXGI Swap Chain, создание буфера цвета, буфера глубины и буфера трафарета. Вторая занимается освобождением всех интерфейсов, которые были получены в первой функции. Рассмотрим их подробнее.

Интерфейсы для работы с DirectX 10

Для полноценной работы с Direct3D-устройством нам понадобится четыре интерфейса:

```
ID3D10Device*                pDevice;
IDXGISwapChain*              pSwapChain;
ID3D10RenderTargetView*      pRenderTargetView;
ID3D10DepthStencilView*      pDepthStencilView;
```

Рассмотрим каждый из них детальнее:

1. *ID3D10Device*. Интерфейс для работы с устройством Direct3D версии 10.0. Это основной интерфейс для работы с 3D-графикой. Он позволяет создавать другие объекты: текстуры, эффекты, вершинные и индексные массивы. При помощи этого интерфейса можно рисовать примитивы и очищать нарисованное. Этот интерфейс занимается установкой настроек рендера.
2. *IDXGISwapChain*. Это интерфейс компонента Swap Chain, который представляет одну или несколько поверхностей/буферов, необходимых для сохранения данных рендеринга, перед тем как они

будут показаны на экране. Он отвечает за установку и настройку буферов цвета, глубины и трафарета. С его помощью происходит переброс информации из back-буфера на front-буфер при двойной буферизации и т.п.

3. *ID3D10RenderTargetView*. Интерфейс буфера вывода. Базируется на RGB 2D-текстуре, иногда в качестве текстуры может выступать back-буфер. В этот буфер производится вывод цветовой части картинки (RGB-значений).
4. *ID3D10DepthStencilView*. Интерфейс для буферов глубины и трафарета. Базируется на 2D-текстуре особого формата, чаще всего это *DXGI_FORMAT_D24_UNORM_S8_UINT*, т.е. в ней на каждый тексель[3] отводится по 24 бита на буфер глубины и по 8 бит на буфер трафарета.

Получение интерфейсов ID3D10Device и IDXGISwapChain

Для получения этих интерфейсов необходимо инициализировать параметры структуры *DXGI_SWAP_CHAIN_DESC* нужными значениями и вызвать функцию *D3D10CreateDeviceAndSwapChain*.

Сначала рассмотрим структуру *DXGI_SWAP_CHAIN_DESC*, которая описывает параметры создаваемого компонента Swap Chain:

```
typedef struct DXGI_SWAP_CHAIN_DESC
{
  DXGI_MODE_DESC   BufferDesc;
  DXGI_SAMPLE_DESC SampleDesc;
  DXGI_USAGE       BufferUsage;
  UINT             BufferCount;
  HWND             OutputWindow;
  BOOL             Windowed;
  DXGI_SWAP_EFFECT SwapEffect;
  UINT             Flags;
} DXGI_SWAP_CHAIN_DESC;
```

Поля структуры:

1. *BufferDesc* – это режим отображения back-буфера. Его ширина и высота, частота обновления и формат пикселя.
2. *SampleDesc* описывает параметры мультисэмплинга. Проще говоря, это параметры полноэкранного сглаживания.

---

[3] Тексел (сокращение от англ. Texture element) – минимальная единица текстуры трёхмерного объекта. Пиксел текстуры.

3. *BufferUsage* – цели использования буфера и настройки доступа к нему из CPU. Back-буфер может использоваться как входной параметр шейдера или как выходной параметр рендера.
4. *BufferCount* – число буферов в Swap Chain, обычно используется один front-буфер для полноэкранного рендера.
5. *OutputWindow* – окно программы, в которое будет производиться вывод.
6. *Windowed* – режим работы 3D-устройства: полноэкранный или оконный. Если данный параметр равен *TRUE*, то будет включен оконный режим.
7. *SwapEffect* определяет, каким образом будет производиться вывод back-буфера на экран. По умолчанию стоит обычное побитовое копирование содержимого back-буфера во front-буфер.
8. *Flags* – дополнительные опции для создания Swap Chain.

Подробную справку по этой структуре и настройках ее параметров можно получить в MSDN Library [3]. Ниже приведен пример ее заполнения для оконного режима:

```
RECT rc = {0}; GetClientRect(hWnd, &rc);

// заполняем структуру необходимыми данными
DXGI_SWAP_CHAIN_DESC dxSCD = {0};
// формат буфера вывода и его размер
dxSCD.BufferCount = 1;
dxSCD.BufferDesc.Width = rc.right;
dxSCD.BufferDesc.Height = rc.bottom;
// этот буфер будет использоваться как буфер вывода на экран
dxSCD.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
// обычный 32-битный цвет для буфера
dxSCD.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
// задаем частоту обновления
dxSCD.BufferDesc.RefreshRate.Numerator = 60;
dxSCD.BufferDesc.RefreshRate.Denominator = 1;
// параметры мультисемплинга (полноэкранного сглаживания картинки)
dxSCD.SampleDesc.Quality = 0;
dxSCD.SampleDesc.Count = 1;
// задаем окно вывода и режим отображения на весь экран
dxSCD.OutputWindow = hWnd;
dxSCD.Windowed = TRUE;

hres = D3D10CreateDeviceAndSwapChain(
      // устройство, отвечающее за вывод (если несколько видеокарт)
      NULL,
      // режим работы устройства (эмуляция, аппаратно и др.)
      D3D10_DRIVER_TYPE_HARDWARE,
      // DLL-модуль, отвечающий за программную эмуляцию
      NULL,
```

```
        // флаги (однопотоковый режим, поддержка BGRA,
        // режим строгой проверки и т.п.)
        D3D10_CREATE_DEVICE_DEBUG,
        // запрашиваемая версия DirectX (должно быть D3D10_SDK_VERSION)
        D3D10_SDK_VERSION,
        &dxSCD,
        // в случае успеха получим интерфейс для Swap Chain
        &pSwapChain,
        // в случае успеха получим интерфейс D3D device
        &pDevice);
if ( FAILED(hres) )
        return FALSE;
```

Как видно из листинга, был получен размер клиентской области окна приложения, заполнены параметры структуры *DXGI_SWAP_CHAIN_DESC* и вызвана функция *D3D10CreateDeviceAndSwapChain* для получения необходимых интерфейсов.

Дадим краткое описание функции *D3D10CreateDeviceAndSwapChain*. Она нужна для получения сразу двух интерфейсов: интерфейса устройства Direct3D и компонента Swap Chain. На вход ей подается заполненная структура *DXGI_SWAP_CHAIN_DESC*, режим работы DirectX 10 и используемая версия SDK. На выходе получаем адреса необходимых интерфейсов. Результат выполнения функции – это стандартный для COM тип *HRESULT*, содержащий код ошибки или *S_OK* в случае успеха.

Получение интерфейса ID3D10RenderTargetView

Следующий этап – создание буфера вывода. Именно на эту поверхность будет производиться рендеринг. В дальнейшем ее сделаем основой для вывода изображения на экран. Буфер вывода будет базироваться на back-буфере из только что созданного Swap Chain.

```
// пытаемся получить бэк-буфер
ID3D10Texture2D* pBackBuffer = 0;
hres = pSwapChain->GetBuffer( 0,
        __uuidof(ID3D10Texture2D),
        (LPVOID*)&pBackBuffer );
if ( FAILED(hres) )
        return FALSE;

// создаем буфер вывода на базе бэк-буфера
hres = pDevice->CreateRenderTargetView(pBackBuffer,
        NULL, &pRenderTargetView);
if ( FAILED(hres) )
        return FALSE;


// бэк-буфер больше не нужен
pBackBuffer->Release();
```

Как видно из вышеприведенного листинга, создание буфера вывода происходит в три этапа:

1. При помощи метода *GetBuffer* мы получаем интерфейс 2D-текстуры нашего back-буфера. Именно на базе этой текстуры и будем создавать буфер вывода.
2. Создается буфер вывода при помощи метода *CreateRenderTargetView* и проверяется успешность выполнения операции.
3. В связи с тем, что интерфейс на 2D-текстуру back-буфера больше не нужен, он освобождается.

Рассмотрим метод GetBuffer подробнее:

```
HRESULT GetBuffer(
      [in]       UINT Buffer,
      [in]       REFIID riid,
      [in, out]  void **ppSurface
);
```

Первый параметр *Buffer* – это индекс буфера, текстуру которого мы запрашиваем. Индексация начинается с нуля. Так как в ранее созданном Swap Chain был всего один back-буфер, то в метод передается 0. Второй параметр *riid* – это идентификатор запрашиваемого интерфейса. С точки зрения технологии COM тип *REFIID* – это просто указатель на *GUID*, иначе говоря, *GUID**. Макрос *__uuidof* позволяет автоматически получить идентификатор у интерфейса. В последний параметр ppSurface будет записан запрашиваемый интерфейс.

Получение интерфейса ID3D10DepthStencilView

Инициализация буфера глубины и буфера трафарета выглядит несколько сложнее. Дело в том, что для этих поверхностей нет готовых текстур, поэтому сначала необходимо создать подходящую текстуру.

```
// сначала создаем для него текстуру
D3D10_TEXTURE2D_DESC dxTD = {0};
dxTD.Width              = rc.right;
dxTD.Height             = rc.bottom;
dxTD.MipLevels          = 1;
dxTD.ArraySize          = 1;
dxTD.Format             = DXGI_FORMAT_D24_UNORM_S8_UINT;
dxTD.SampleDesc.Count   = 1;
dxTD.SampleDesc.Quality = 0;
dxTD.Usage              = D3D10_USAGE_DEFAULT;
```

```
dxTD.BindFlags          = D3D10_BIND_DEPTH_STENCIL;
dxTD.CPUAccessFlags     = 0;
dxTD.MiscFlags          = 0;

ID3D10Texture2D *pDepthStencil = 0;
hres = pDevice->CreateTexture2D(&dxTD, 0, &pDepthStencil);
if ( FAILED(hres) )
      return FALSE;
```

Создание текстуры происходит в два этапа: сначала заполняется структура *D3D10_TEXTURE2D_DESC*, описывающая параметры создаваемой текстуры, а затем вызывается метод *CreateTexture2D*, который завершает создание. Рассмотрим *D3D10_TEXTURE2D_DESC* подробнее:

```
typedef struct D3D10_TEXTURE2D_DESC {
  UINT              Width;
  UINT              Height;
  UINT              MipLevels;
  UINT              ArraySize;
  DXGI_FORMAT       Format;
  DXGI_SAMPLE_DESC  SampleDesc;
  D3D10_USAGE       Usage;
  UINT              BindFlags;
  UINT              CPUAccessFlags;
  UINT              MiscFlags;
} D3D10_TEXTURE2D_DESC;
```

1. Параметры *Width* и *Height* – это соответственно ширина и высота создаваемой текстуры. Размер нашей текстуры равен размеру клиентской области, который, в свою очередь, равен размеру back-буфера. Таким образом, буфер вывода, буфер глубины и буфер трафарета будут одного размера.
2. *MipLevels* – это количество субтекстур, также называемых mipmap-уровнями. Если указать 1, то будет создана мультисэмпл-текстура, если указать 0, то будет сгенерирован весь набор субтекстур.
3. *ArraySize* указывает число текстур в текстурном массиве. В нашем случае нужна всего одна текстура.
4. *Format* – это формат запрашиваемой текстуры. *DXGI_FORMAT_D24_UNORM_S8_UINT* – это 32-битный формат, в котором на буфер глубины отводится 24 бита, а на буфер трафарета – 8 бит. Если буфер трафарета не используется, то можно запросить текстуру *DXGI_FORMAT_R24_UNORM_X8_TYPELESS* или *DXGI_FORMAT_D16_UNORM*.

5. Параметр *SampleDesc* описывает параметры мультисэмплинга создаваемой текстуры. В нашем случае мультисэмплинг «отключен», т.е. на каждый тексель текстуры приходится по одному сэмплу.
6. Параметр *Usage* отвечает за то, каким образом будет использоваться текстура: доступна только из GPU, доступна и CPU, и GPU или нужна для копирования из GPU в CPU. По умолчанию предоставляется доступ на чтение и запись только для GPU.
7. *BindFlags* – флаги, отвечающие за назначение создаваемой текстуры, т.е. на каком этапе рендеринга она будет использоваться.
8. Флаги *CPUAccessFlags* и *MiscFlags* нужны для дополнительной информации о создаваемой текстуре: уровне доступа от CPU, совместимости с GDI-устройствами и т.п.

Создание текстуры происходит при помощи метода *CreateTexture2D*. Он вернет интерфейс на только что созданную 2D-текстуру.

Для завершения инициализации буферов трафарета и глубины необходимо заполнить структуру *D3D10_DEPTH_STENCIL_VIEW_DESC* и вызвать метод *CreateDepthStencilView*:

```
// теперь на базе текстуры создаем буфер
D3D10_DEPTH_STENCIL_VIEW_DESC dxDSVD = {0};
dxDSVD.Format = dxTD.Format;
dxDSVD.ViewDimension = D3D10_DSV_DIMENSION_TEXTURE2D;
hres = pDevice->CreateDepthStencilView(pDepthStencil,
      &dxDSVD, &pDepthStencilView);
if ( FAILED(hres) )
      return FALSE;

// освобождаем depth-текстуру
pDepthStencil->Release();
```

Рассмотрим подробнее эти шаги. Структура *D3D10_DEPTH_STENCIL_VIEW_DESC* определяет, как получить доступ к ресурсу, используемому в качестве буферов глубины и трафарета.

```
typedef struct D3D10_DEPTH_STENCIL_VIEW_DESC {
  DXGI_FORMAT         Format;
  D3D10_DSV_DIMENSION ViewDimension;
  union {
    D3D10_TEX1D_DSV         Texture1D;
    D3D10_TEX1D_ARRAY_DSV   Texture1DArray;
    D3D10_TEX2D_DSV         Texture2D;
    D3D10_TEX2D_ARRAY_DSV   Texture2DArray;
```

```
    D3D10_TEX2DMS_DSV       Texture2DMS;
    D3D10_TEX2DMS_ARRAY_DSV Texture2DMSArray;
  };
} D3D10_DEPTH_STENCIL_VIEW_DESC;
```

В этой структуре надо заполнить только параметры *Format* и *ViewDimension*. Первый – копия формата только что созданной 2D-текстуры. Второй параметр указывает, в каком виде эта текстура будет доступна, в нашем случае как обычная 2D-текстура.

На вход методу *CreateDepthStencilView* подается depth-stencil-текстура и заполненная структура *D3D10_DEPTH_STENCIL_VIEW_DESC*. На выходе будет интерфейс на буферы глубины и трафарета. Так как в дальнейшем интерфейс созданной в этом параграфе 2D-текстуры не понадобится, то мы его освобождаем. Сама текстура при этом не уничтожается – ссылка на нее хранится в только что созданном буфере данных.

Установка буферов вывода, трафарета и глубины

После того как получены все необходимые интерфейсы, их можно установить в качестве целей рендера. Для этого используется метод *OMSetRenderTargets*:

```
pDevice->OMSetRenderTargets(1,
      &pRenderTargetView, pDepthStencilView);
```

В качестве буфера вывода можно установить сразу несколько целей-поверхностей. За их количество отвечает первый параметр. Остальные параметры, очевидно, представляют собой указатели на интерфейсы буферов.

На этом инициализация DirectX 10 закончена. Можно загружать ресурсы и производить отрисовку.

Освобождение интерфейсов DirectX 10

По завершении работы программы контекст DirectX нужно освободить, предварительно убедившись, что все ресурсы приложения выгружены из памяти. Сама очистка контекста выглядит невероятно просто:

```
SAFE_RELEASE(pDepthStencilView);
SAFE_RELEASE(pRenderTargetView);
SAFE_RELEASE(pSwapChain);
SAFE_RELEASE(pDevice);
```

Как видно, мы просто по очереди освобождаем все четыре запрошенных интерфейса. Порядок освобождения обратный порядку создания. Для упрощения кода используется специальный макрос *SAFE_RELEASE*, выглядит он так:

```
#define SAFE_RELEASE(p) { if (p) p->Release(); p = 0; }
```

Этот макрос вызывает метод Release, своего параметра, а затем присваивает параметру значение 0. Таким образом, в дальнейшем этот параметр будет невозможно использовать.

ИНИЦИАЛИЗАЦИЯ И ОСВОБОЖДЕНИЕ ГРАФИЧЕСКИХ РЕСУРСОВ

После того как создан и инициализирован DirectX, можно загружать ресурсы, необходимые в процессе рендеринга. Нужно помнить, что загрузка должна происходить только один раз! Это связано с тем, что время обращения к винчестеру несоизмеримо больше, чем время обращения к оперативной памяти или видеопамяти. Современные видеокарты обладают большими объемами памяти, что позволяет сразу загрузить все необходимые ресурсы и в дальнейшем не тратить время на дополнительные дозагрузки в процессе рендеринга. Создавать ресурсы в процессе рендеринга нужно только в самых крайних случаях, когда другого варианта не остается.

Итак, после успешного создания DirectX-контекста можно начинать создавать необходимые ресурсы. В примере это будут:

- эффект, содержащий вершинный и фрагментный шейдеры;
- вершинный и индексный буферы, содержащие описание 3D-модели куба;
- текстура, накладываемая на куб.

Инициализация каждого ресурса выглядит достаточно громоздко. Это связано с тем, что Direct3D предоставляет множество возможностей использования одного и того же ресурса. Например, текстура может быть доступна не только для чтения, но и для записи; в текстуру можно загружать изображение из файла или рисовать непосредственно в нее и т.п. Разработчик должен помнить, что вспомогательные функции и классы, напи-

санные как обертка над DirectX-вызовами, существенно облегчат дальнейшее создание программы.

Интерфейсы, необходимые для управления ресурсами, выглядят так:

```
ID3D10Effect*           pEffect;
ID3D10EffectTechnique*  pTechnique;
ID3D10InputLayout*      pInputLayout;
ID3D10Buffer*           pVertexBuffer;
ID3D10Buffer*           pIndexBuffer;
```

Ниже будет детально рассмотрена инициализация каждого ресурса.

Загрузка эффекта и получение необходимой техники

Подробно о том, что такое эффект и техника, можно прочитать в методическом пособии [2]. Исходный код эффекта есть в примере и в приложении 2. В этом параграфе лишь отметим, что эффект представляет собой текстовый файл, содержащий код шейдеров на языке HLSL. И как любой другой код, перед использованием его нужно скомпилировать. Разница с обычным кодом, например на языке C++, заключается в том, что сборка происходит на машине пользователя, а не на машине разработчика. Поэтому DirectX содержит внутри полноценный компилятор. Драйвер собирает HLSL-код в наиболее оптимальный вариант именно для видеокарты пользователя, что позволяет добиться максимальной производительности на любой конфигурации компьютера.

Сам код загрузки выглядит таким образом:

```
ID3D10Blob* pErrors = 0;
hres = D3DX10CreateEffectFromFile(
      TEXT("simple.fx"),           // имя файла
      NULL, NULL,
      "fx_4_0",
      D3D10_SHADER_ENABLE_STRICTNESS, 0,
      pDevice, NULL, NULL, &pEffect, &pErrors,
      NULL);
if ( FAILED(hres) )
{
      if (pErrors)
      {
            OutputDebugStringA( (char*)pErrors->GetBufferPointer() );
            SAFE_RELEASE(pErrors);
      }
      return FALSE;
}
```

Он выполняется при помощи функции *D3DX10CreateEffectFromFile*. Ее прототип выглядит так:

```
HRESULT D3DX10CreateEffectFromFile(
  __in   LPCTSTR pFileName,
  __in   const D3D10_SHADER_MACRO *pDefines,
  __in   ID3D10Include *pInclude,
  __in   LPCSTR pProfile,
  __in   UINT HLSLFlags,
  __in   UINT FXFlags,
  __in   ID3D10Device *pDevice,
  __in   ID3D10EffectPool *pEffectPool,
  __in   ID3DX10ThreadPump *pPump,
  __out  ID3D10Effect **ppEffect,
  __out  ID3D10Blob **ppErrors,
  __out  HRESULT *pHResult
);
```

1. Параметр *pFileName* отвечает за имя текстового файла, содержащего текст эффекта.
2. *pDefines* – массив, содержащий множество пар строк, первая строка отвечает за имя макроса, вторая строка в паре – его значение. Массив заканчивается нулем. Таким образом, макросы можно указывать просто как массив строк, в котором каждая строка завершается нулем, а сам массив завершается двумя нулевыми символами. Дополнительных макросов для эффекта не нужно, поэтому заносим *NULL*, указывающий на отсутствие макросов.
3. *pInclude* – это указатель на интерфейс *ID3D10Include*, он позволяет приложению создавать перегружаемые методы для открытия и закрытия файлов, если эффект загружается из памяти. Так как наш эффект загружается из файла на диске, то указывается *NULL*.
4. Параметр *pProfile* отвечает за версию шейдерной модели. Например, Direct3D 9 поддерживает профили начиная от версии 1.1 и до 3.0, Direct3D 10 – от 4.0 до 4.1 (для версии D3D10.1), Direct3D 11 – от 4.0 до 5.0. Запись вида *fx_4_0* означает, что используется эффект, содержащий шейдеры версии 4.0.
5. Параметр *HLSLFlags* содержит опции для компиляции HLSL-шейдера. В примере указан только один флаг, отвечающий за строгость компилятора к устаревшему синтаксису. Однако можно указать необходимость в отладочной информации или, наоборот, максимальной оптимизации кода, потребовать обратную совместимость кода и т.п.

6. *FXFlags* – это опции для компиляции эффекта. В нашем примере никаких дополнительных опций не требуется.
7. *pDevice* – интерфейс на контекст Direct3D, для которого будет собираться шейдер.
8. *pEffectPool* – указатель на интерфейс *ID3D10EffectPool* общей области памяти, отвечающей за совместное использование переменных для разных эффектов. Не используется.
9. *pPump* – указатель на интерфейс *ID3DX10ThreadPump*, который необходим для асинхронной загрузки ресурсов. Если мы указываем NULL, то функция не вернет управление, пока загрузка не завершится. То есть загрузка будет происходить в том же самом потоке.
10. *ppEffect*, в этот параметр будет занесен указатель на созданный эффект в случае успешной компиляции и сборки шейдеров.
11. *ppErrors*, в этот параметр можно передавать NULL, но если интересует подробный результат компиляции эффекта, описание ошибок и другая информация, то надо задать указатель на интерфейс *ID3D10Blob*. Описание ошибок предоставляется в удобочитаемом для человека виде, т.е. в виде нормального текста с описанием места и причины ошибки.
12. *pHResult* нужен для асинхронной загрузки ресурсов. В этот параметр будет записан результат загрузки и компиляции эффекта. Программист обязан гарантировать существование этого параметра на протяжении всего времени загрузки шейдера.

По окончании выполнения функции проверяется успешность загрузки и компиляции эффекта. В случае ошибки используется интерфейс *ID3D10Blob* и в отладочное окно Visual Studio будет занесен подробный отчет о причинах сбоя. В случае успеха в переменной *pEffect* будет указатель на интерфейс загруженного эффекта.

Следующий этап – это получение техники и указателей на глобальные переменные шейдера. Они будут использоваться в дальнейшем процессе рендеринга. Все эти параметры запрашиваются у интерфейса эффекта и будут существовать, пока существует эффект.

```
// Получаем технику
pTechnique = pEffect->GetTechniqueByName("tech0");



// Получаем глобальные переменные шейдера
pWVPVariable = pEffect->GetVariableByName("wvp")->AsMatrix();
pWorldInverseVariable = pEffect->GetVariableByName("wi")->AsMatrix();
pDiffuseVariable = pEffect->GetVariableByName("texDiffuse")
                          ->AsShaderResource();
pLightVariable = pEffect->GetVariableByName("light")->AsVector();
```

Параметры запрашиваются по имени, которым они обозначены в файле эффекта. Для глобальных переменных нужно уточнить, в каком формате они поступают к нам в программу. То есть для благополучия и долгой жизни вашей программы не стоит запрашивать вектор как матрицу.

Занесение данных о 3D-модели в видеопамять

В начале своего существования видеокарты работали с оперативной памятью компьютера, что не самым лучшим образом сказывалось на производительности. Сейчас для достижения пиковой производительности все данные, с которыми будет работать GPU, должны находиться в собственной памяти видеокарты. Поэтому в новых версиях DirectX полностью отсутствует возможность отрисовки моделей, находящихся в оперативной памяти. Всю информацию нужно записывать в видеопамять. Эта операция производится в три этапа: настройка формата вершины, запись вершинных данных в видеопамять и запись индексных данных в видеопамять. Рассмотрим каждый этап подробно.

В связи с тем, что для разных задач вершина 3D-модели может содержать разную информацию, необходимо указывать ее формат, перед тем как заносить ее в видеопамять. Например, для простого шейдера будет достаточно текстурных координат, нормали и позиции для каждой вершины модели. Но как только мы попытаемся реализовать «тяжелый» эффект, обнаружим, что этой информации недостаточно и нужно указывать бинормаль, тангент, дополнительные текстурные координаты, цвет, вес костей и т.п. Возникает вопрос: почему сразу не воспользоваться форматом вершины, содержащим избыточную информацию? Ответ прост: память не резиновая. И выхода у программиста два: или он сможет уложить все необходимые данные в минимальные системные требования, или программа будет безбожно тормозить и спотыкаться, если вообще запустится.

Итак, настройки формата вершины для шейдера выглядят следующим образом:

```
// описание вершины
D3D10_INPUT_ELEMENT_DESC dxIED[] =
{
    { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 0,
        D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 8,
        D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 8 + 12,
        D3D10_INPUT_PER_VERTEX_DATA, 0 },
};
UINT numElements = ARRAY_SIZE(dxIED);

// создаем input layout (нужен для сопоставления входных
// данных вершинного шейдера и формата вершины)
D3D10_PASS_DESC dxPD = {0};
pTechnique->GetPassByName("p0")->GetDesc(&dxPD);
hres = pDevice->CreateInputLayout(
    dxIED,                    // массив с описанием вершины
    numElements,              // кол-во элементов в этом массиве
    dxPD.pIAInputSignature,   // указатель на скомпилированный шейдер
    dxPD.IAInputSignatureSize, // размер скомпилированного шейдера
    &pInputLayout);
if( FAILED( hres ) )
    return FALSE;
```

Сначала заполняется массив структур *D3D10_INPUT_ELEMENT_DESC*, каждый элемент которого представляет один из параметров вершины (позицию, нормаль или что-то еще). Затем, получив информацию о шейдере, сопоставляем его с форматом вершины при помощи метода *CreateInputLayout*. Рассмотрим этот код подробнее.

Структура *D3D10_INPUT_ELEMENT_DESC* выглядит следующим образом:

```
typedef struct D3D10_INPUT_ELEMENT_DESC {
  LPCSTR                    SemanticName;
  UINT                      SemanticIndex;
  DXGI_FORMAT               Format;
  UINT                      InputSlot;
  UINT                      AlignedByteOffset;
  D3D10_INPUT_CLASSIFICATION InputSlotClass;
  UINT                      InstanceDataStepRate;
} D3D10_INPUT_ELEMENT_DESC;
```

1. *SemanticName* – имя HLSL-семантики, ассоциированное с данным элементом. Подробнее обо всех видах семантик можно прочитать в MSDN Library, ими могут быть: POSITION (позиция), NORMAL

(нормаль), COLOR (цвет), TEXCOORD (текстурная координата), TANGENT (тангент) и другие параметры вершины.

2. *SemanticIndex* – индекс семантики у элемента. Этот индекс необходим, только если в описании вершины существует больше чем один элемент с одинаковым именем у семантики. Например, матрица 4×4 будет иметь четыре компонента, каждый с семантическим именем «MATRIX», но с разными индексами: 0, 1, 2 и 3.
3. *Format* – тип данных для этого элемента. Подробнее о возможных типах рассказано в описании формата *DXGI_FORMAT*. В примере используются форматы, содержащие по два или три компонента типа *float*. Например, *DXGI_FORMAT_R32G32B32_FLOAT* представляет собой вектор из трех компонент типа *float* (по 32 бита на каждый), в котором R-, G- и B-компоненты отвечают соответственно за X, Y и Z.
4. *InputSlot* – целочисленное значение, которое идентифицирует входной слот. Допустимые значения – от 0 до 15. DirectX позволяет при рендеринге указывать несколько входных вершинных массивов. Например, текстурные координаты и позиция в одном массиве, а нормали – в другом. Поэтому при описании формата вершины необходимо указывать, из какого именно массива брать тот или иной параметр.
5. *AlignedByteOffset* – смещение в байтах до первого вхождения этого элемента. В нашем случае самым первым элементом в вершине является текстурная координата (поэтому смещение до нее равно 0), затем идет нормаль (смещение до нее равно размеру текстурной координаты, т.е. двум float'ам или 8 байтам), а затем – позиция (смещение равно сумме размеров текстурной координаты и нормали). Если элементы в вершине идут один за другим, без разрывов и дополнительных смещений, то можно воспользоваться вместо числового значения параметром *D3D10_APPEND_ALIGNED_ELEMENT*. В этом случае смещение будет рассчитано автоматически, исходя из типов элементов вершины.
6. *InputSlotClass* – указывает класс входных данных для каждого входного слота. Может принимать два значения:

D3D10_INPUT_PER_VERTEX_DATA
или *D3D10_INPUT_PER_INSTANCE_DATA*, соответственно повершинный или пообъектный. То есть второй параметр применяется не к отдельной вершине, а к целому объекту (если в пределах одного массива вершин их упаковано несколько).

7. *InstanceDataStepRate* – в случае если *InputSlotClass* установлен в *D3D10_INPUT_PER_INSTANCE_DATA*, данный параметр указывает, сколько нужно пропустить (в смысле нарисовать) вершин, прежде чем мы перейдем к следующему объекту. Параметры 6 и 7 нужны для так называемой технологии instacing'а, когда за один вызов можно нарисовать огромное число одинаковых (и не очень) объектов, например, лесной массив или орду вражеских войск.

После того как массив структур заполнен, можно начинать сопоставление формата вершины и соответствующего ей прохода (PASS) у эффекта. Для этого при помощи метода *GetDesc* получаем описание прохода у нашего эффекта. И при помощи метода *CreateInputLayout* создаем экземпляр интерфейса pInputLayout, который отвечает за это сопоставление. Прототип данного метода выглядит так:

```
HRESULT CreateInputLayout(
  [in]   const D3D10_INPUT_ELEMENT_DESC *pInputElementDescs,
  [in]   UINT NumElements,
  [in]   const void *pShaderBytecodeWithInputSignature,
  [in]   SIZE_T BytecodeLength,
  [out]  ID3D10InputLayout **ppInputLayout
);
```

1. *pInputElementDescs* – массив структур *D3D10_INPUT_ELEMENT_DESC*, который мы заполняли ранее.
2. *NumElements* – количество элементов в этом массиве.
3. *pShaderBytecodeWithInputSignature* – указатель на скомпилированный шейдер (эффект). Этот указатель в примере получен из описания прохода эффекта.
4. *BytecodeLength* – размер скомпилированного шейдера в байтах. Этот параметр также можно получить у прохода эффекта.
5. *ppInputLayout* – указатель на интерфейс, в который будет занесен созданный экземпляр *ID3D10InputLayout*.

Следующим этапом записи 3D-модели в видеопамять будет создание ее вершинного массива, т.е. массива, содержащего информацию о каждой вершине в модели. Выглядит этот этап так:

```
// описание буфера и его предназначение
D3D10_BUFFER_DESC dxBD = {0};
dxBD.ByteWidth = sizeof(meshV);          // размер буфера в байтах
dxBD.Usage = D3D10_USAGE_DEFAULT;        // как будет использоваться
                                         // (чтение/запись)
dxBD.BindFlags = D3D10_BIND_VERTEX_BUFFER;     // в каком качестве буфер
                                               // будет использоваться
                                               // (как вершинные данные)
dxBD.CPUAccessFlags = 0;                 // доступ к буферу для процессора
                                         // (0, если нужен только для
                                         // видеочипа)
dxBD.MiscFlags = 0;

// содержимое буфера
D3D10_SUBRESOURCE_DATA dxSD = {0};
dxSD.pSysMem = meshV;                    // массив данных
dxSD.SysMemPitch = 0;                    // для текстурирования
                                         // (pitch по ширине)
dxSD.SysMemSlicePitch = 0;               // для текстурирования
                                         // (pitch по глубине)

// создаем буфер и заносим в видеопамять
hres = pDevice->CreateBuffer(&dxBD, &dxSD, &pVertexBuffer);
if( FAILED( hres ) )
      return FALSE;
```

В этом примере есть одна неизвестная переменная – *meshV*. Это массив, содержащий описание каждой вершины. Подробное описание модели приведено в приложении 1 и в исходном коде примера. Для записи вершинного массива в видеопамять нужно заполнить две структуры: *D3D10_BUFFER_DESC* и *D3D10_SUBRESOURCE_DATA*. Первая отвечает за описание буфера в видеопамяти и его назначение (так как в видеопамять можно записывать еще и текстуры, индексные массивы и т.п.), а вторая – за содержимое этого буфера.

Структура *D3D10_BUFFER_DESC* выглядит так:

```
typedef struct D3D10_BUFFER_DESC {
  UINT        ByteWidth;
  D3D10_USAGE Usage;
  UINT        BindFlags;
  UINT        CPUAccessFlags;
  UINT        MiscFlags;
} D3D10_BUFFER_DESC;
```

1. *ByteWidth* – размер буфера в байтах.
2. *Usage* – указывает на то, как буфер будет использоваться (читаться и записываться). Частота обновления содержимого буфера – ключевой фактор. Наиболее часто используемое значение – *D3D10_USAGE_DEFAULT*. Остальные параметры можно посмотреть в справке по перечислению *D3D10_USAGE*.
3. *BindFlags* – указывает на то, в каком качестве будет использоваться буфер. Маска составляется при помощи «побитового или». Подробно о составляющих битовой маски рассказано в описании перечисления *D3D10_BIND_FLAG*. В нашем случае буфер используется как вершинный.
4. *CPUAccessFlags* – данные флаги указывают на то, какой доступ у центрального процессора будет к этому блоку в видеопамяти, или ноль, если CPU не будет иметь доступа вообще. Флаги можно объединять при помощи «побитового или». Доступные флаги: *D3D10_CPU_ACCESS_WRITE* (CPU имеет доступ на запись в этот буфер) и *D3D10_CPU_ACCESS_READ* (CPU имеет доступ на чтение из этого буфера).
5. *MiscFlags* – остальные флаги, они перечислены в справке к *D3D10_RESOURCE_MISC_FLAG*. В нашем случае указан ноль, что означает «никакие флаги не установлены». Флаги можно комбинировать при помощи «побитового или».

Следующая структура – *D3D10_SUBRESOURCE_DATA*, которая содержит всего три поля:

```
typedef struct D3D10_SUBRESOURCE_DATA {
  const void *pSysMem;
  UINT       SysMemPitch;
  UINT       SysMemSlicePitch;
} D3D10_SUBRESOURCE_DATA;
```

1. *pSysMem* – указатель на данные в оперативной памяти.
2. *SysMemPitch* – расстояние в байтах от начала одной линии в текстуре до начала следующей. В нашем случае не используется и равно нулю. Это означает, что в памяти данные лежат непрерывно.

3. *SysMemSlicePitch* – расстояние в байтах от начала одного уровня глубины и до начала следующего. Используется только в 3D-текстурах, поэтому в нашем примере это поле равно нулю.

После того как структуры заполнены, вызывается метод *CreateBuffer*, который выделяет необходимую область в видеопамяти, копирует в нее информацию из оперативной памяти и затем возвращает указатель на соответствующий интерфейс (прямого доступа к видеопамяти, естественно, вы получить не сможете, точнее, сможете, но только на консолях, которые не рассматриваются в данном учебно-методическом пособии).

По аналогии создается индексный массив 3D-модели. Код приведен ниже.

```
// описание буфера и его предназначение
dxBD.ByteWidth = sizeof(meshI);           // размер буфера в байтах
dxBD.Usage = D3D10_USAGE_DEFAULT;         // как будет использоваться
                                          // (чтение/запись)
dxBD.BindFlags = D3D10_BIND_INDEX_BUFFER;      // в каком качестве буфер
                                               // будет использоваться
                                               // (как индексные данные)
dxBD.CPUAccessFlags = 0;                  // доступ к буферу для процессора
                                          // (0, если нужен только для
                                          // видеочипа)
dxBD.MiscFlags = 0;

// содержимое буфера
dxSD.pSysMem = meshI;                     // массив данных
dxSD.SysMemPitch = 0;                     // для текстурирования
                                          // (pitch по ширине)
dxSD.SysMemSlicePitch = 0;                // для текстурирования
                                          // (pitch по глубине)

// создаем буфер и заносим в видеопамять
hres = pDevice->CreateBuffer(&dxBD, &dxSD, &pIndexBuffer);
if( FAILED( hres ) )
      return FALSE;
```

Переменная *meshI* представляет собой массив целых чисел, описывающих последовательность передачи вершин модели на растеризатор, это называется массивом индексов. Код практически идентичен созданию буфера вершин. Единственное отличие – поле *BindFlags* в структуре *D3D10_BUFFER_DESC*, в нем указано другое назначение буфера (индексный буфер вместо вершинного). И, естественно, использовали новый указатель в оперативной памяти и размер буфера (*meshI* вместо *meshV*).

Загрузка текстуры

Осталось загрузить последний ресурс – текстуру. Это самый простой этап, так как в DirectX есть встроенный загрузчик некоторых графических форматов файлов, в частности, Direct Draw Surface (DDS). Итак, код загрузки выглядит так:

```
hres = D3DX10CreateShaderResourceViewFromFile(
        pDevice,
        TEXT("simple.dds"),
        NULL, // D3DX10_IMAGE_LOAD_INFO* - информация о загруженной текстуре
        NULL, // ID3DX10ThreadPump interface (многопоточная загрузка)
        &pTexture,
        NULL);      // результат операции (если используется
                    // многопотоковый ID3DX10ThreadPump)
if( FAILED(hres) )
    return FALSE;
```

Одна из новых возможностей DirectX 10 – асинхронная загрузка ресурсов. Это означает, что мы можем запускать приложение, не дождавшись загрузки всех ресурсов, и они просто появятся позднее. С учетом того, что сейчас многие процессоры содержат несколько ядер (на которые можно переложить загрузку), подобный подход значительно повышает скорость запуска приложения. Рассмотрим подробнее функцию загрузки:

```
HRESULT D3DX10CreateShaderResourceViewFromFile(
    ID3D10Device *pDevice,
    LPCTSTR pSrcFile,
    D3DX10_IMAGE_LOAD_INFO *pLoadInfo,
    ID3DX10ThreadPump *pPump,
    ID3D10ShaderResourceView **ppShaderResourceView,
    HRESULT *pHResult
);
```

1. *pDevice* – интерфейс нашего устройства Direct3D 10.
2. *pSrcFile* – имя файла загружаемого ресурса, обычная C-строка, оканчивающаяся нулем. Следите за настройками компилятора, в случае Unicode надо передавать строки, содержащие wchar_t-символы.
3. *pLoadInfo* – опциональный параметр, можно не указывать (NULL). Это указатель на структуру *D3DX10_IMAGE_LOAD_INFO*, в которой хранится дополнительная информация о загружаемом файле: ширина, высота, число мипмап-уровней, формат пикселя и т.п. Если указан NULL, то информация о текстуре будет браться из самого файла.

4. *pPump* – интерфейс для асинхронной загрузки изображения. Если вместо него вписываем 0, то загрузка будет обычной и функция не вернет управление до тех пор, пока не загрузит текстуру.
5. *ppShaderResourceView* – сюда будет записан адрес интерфейса для загруженной текстуры.
6. *pHResult* – результат выполнения асинхронной загрузки. Нужно обязательно указывать, если pPump не нулевой. В противном случае можно вписать 0.

На этом загрузка всех необходимых ресурсов закончена. Таким образом, для тестового приложения, рисующего текстурированный 3D-кубик, понадобилось всего четыре ресурса: *вершинный* и *индексный буферы*, которые определяют геометрию модели; *текстура*, определяющая фактуру модели; и *эффект*, занимающийся расчетами освещения, позиции и наложения текстуры.

Освобождение загруженных ресурсов в конце работы программы

Аналогично освобождению устройств DirectX освобождение ресурсов выглядит тоже просто:

```
SAFE_RELEASE(pTexture);
SAFE_RELEASE(pIndexBuffer);
SAFE_RELEASE(pVertexBuffer);
SAFE_RELEASE(pInputLayout);
SAFE_RELEASE(pEffect);
```

Мы выгружаем ресурсы в порядке, обратном порядку загрузки. Очисткой видеопамяти будет заниматься драйвер DirectX.

ПРОЦЕСС РЕНДЕРИНГА ИЗОБРАЖЕНИЯ

В цикле обработки сообщений приложения непрерывно вызывается функция Render. Она отвечает за генерацию картинки, которую мы увидим на экране. Функция состоит из пяти этапов:

1. Очистка экрана.
2. Настройка камеры и позиции источников света.
3. Установка параметров эффекта (шейдера) и текущего прохода.
4. Отрисовка модели (меша).
5. Вывод получившейся картинки на экран.

Также в примере выполняются и другие действия: проверки на правильность созданных ресурсов и устройств DirectX, вычисляется время, прошедшее с момента рендеринга предыдущего кадра и т.п. В учебно-методическом пособии это не рассматривается, сконцентрируемся только на основных этапах.

Очистка экрана

В большинстве приложений рендер начинается с очистки экрана.

```
pDevice->ClearRenderTargetView(pRenderTargetView,
      D3DXCOLOR(0.0f, 0.125f, 0.155f, 1.0f));
pDevice->ClearDepthStencilView(pDepthStencilView,
      D3D10_CLEAR_DEPTH | D3D10_CLEAR_STENCIL, 1.0f, 0);
```

В 3D-приложении недостаточно просто очистить видимую часть экрана. Очистке подвергаются дополнительные битовые плоскости (подробнее о рендеринге 3D-изображения рассказано в первом методическом пособии), поэтому для очистки экрана вызывается два метода:

- *ClearRenderTargetView* – для очистки видимой части экрана. В качестве первого параметра ему передается интерфейс на буфер вывода, так как именно в него и производится вывод видимой части сцены. В качестве второго параметра – цвет очистки, для указания которого используется макрос D3DXCOLOR. Макрос принимает четыре параметра: красный, зеленый, синий и альфа-компоненты. Допустимые значения – от 0 до 1.
- *ClearDepthStencilView* – для очистки невидимых битовых плоскостей: глубины и трафарета. В качестве первого параметра передается соответствующий интерфейс на буфер трафарета и глубины. Во втором параметре при помощи битовой маски указывается, какой из буферов будет подвергаться очистке (в нашем случае оба). Третий и четвертый параметры – соответственно значения по умолчанию для каждого текселя в буфере глубины и буфере трафарета.

Стоит понимать, что процесс очистки экрана занимает длительное время (с точки зрения отрисовки одного кадра), поэтому без особой необходимости не стоит очищать ненужные буферы. Например, если при рендеринге изображения новый кадр затирает весь старый кадр, то можно не очищать буфер вывода и сэкономить несколько FPS.

Настройка камеры

Следующий этап – подготовка проекционной, видовой и мировой матриц. Подробный рассказ о матричных преобразованиях приведен в методическом пособии [1]. Матрицы можно подготовить самостоятельно или при помощи методов DirectX. Также стоит отметить, что сами матрицы нужны только внутри шейдера. С точки зрения DirectX матрица – это просто один из параметров шейдера, такой же как текстура или константа.

Для подготовки матриц будем использовать возможности DirectX. Понадобится три матрицы, соответственно мировая, видовая и проекционная:

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

Все они типа *D3DXMATRIX*, который представляет собой структуру, содержащую 16 чисел типа float.

Проекционная матрица задается при помощи функции *D3DXMatrixPerspectiveFovLH*:

```
D3DXMatrixPerspectiveFovLH(
      &mProjection,
      (FLOAT)D3DX_PI * 0.5f,
      iWidth / (FLOAT)iHeight,
      0.1f, 100.0f);
```

Ее объявление выглядит следующим образом:

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

1. *pOut* – указатель на матрицу, в которую будет занесен результат.
2. *fovy* – угол обзора по оси Y в радианах (в примере 90°).

3. *Aspect* – соотношение ширины окна к ее высоте. Этот параметр нужен для того, чтобы при изменении размеров окна сохранялись пропорции у объектов.
4. *zn*, *zf* – соответственно ближняя и дальняя плоскости отсечения. Ближняя не может быть меньше или равна нулю. Большое расстояние от ближней плоскости отсечения до дальней плоскости может привести к неприятным артефактам в изображении.

Видовая матрица – это позиция камеры и направление ее «взгляда». Она задается при помощи функции *D3DXMatrixLookAtLH*.

```
D3DXVECTOR3 eye( 0.0f, 1.0f, -4.0f );
D3DXVECTOR3 at( 0.0f, 0.0f, 0.0f );
D3DXVECTOR3 up( 0.0f, 1.0f, 0.0f );
D3DXMatrixLookAtLH( &mView, &eye, &at, &up );
```

Прототип функции выглядит так:

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

1. *pOut* – указатель на матрицу, в которую будет записан результат.
2. *pEye* – указатель на объект структуры типа *D3DXVECTOR3* (обычный трехмерный вектор: x, y, z). В этот вектор записано положение камеры.
3. *pAt* – точка в пространстве, на которую направлена камера.
4. *pUp* – вектор, определяющий «верх» для камеры, обычно это вектор вида (0, 1, 0). Данный вектор должен быть нормализован.

И последняя матрица – мировая. Она задает положение текущего объекта в пространстве. Каких-то специальных функций для ее инициализации нет. В примере объект-куб просто вращается вокруг своей оси, поэтому код для вычисления его позиции выглядит так:

```
D3DXVECTOR3 axis(1, 1, 0);
D3DXMatrixIdentity(&mWorld);
D3DXMatrixRotationAxis(&mWorld, &axis, t / 5);
```

Сначала при помощи функции *D3DXMatrixIdentity* матрица вида становится единичной. Затем при помощи *D3DXMatrixRotationAxis* она преобразуется в матрицу поворота вокруг некоторой оси. Второй параметр этой функции – ось поворота. Третий параметр – угол в радианах (в этом примере параметр t плавно увеличивается от нулевого значения при старте программы).

В эффекте не используются эти матрицы в «чистом» виде. В него необходимо передать всего две матрицы: результат произведения всех трех матриц и инвертированную мировую матрицу.

```
// WorldViewProjection
D3DXMATRIX wvp;
D3DXMatrixMultiply(&wvp, &mWorld, &mView);
D3DXMatrixMultiply(&wvp, &wvp, &mProjection);
// WorldInverse
D3DXMATRIX wi;
D3DXMatrixInverse(&wi, 0, &mWorld);
```

В этом коде стоит пояснить только второй параметр у функции *D3DXMatrixInverse*. Это указатель на число с плавающей точкой, в котором записан детерминант матрицы. Если вместо него установлен 0, то детерминант вычисляется автоматически.

Настройка переменных эффекта

В конце главы «Загрузка эффекта и получение необходимой техники» были проинициализированы следующие интерфейсы: *pWVPVariable*, *pWorldInverseVariable*, *pDiffuseVariable* и *pLightVariable*. Именно с их помощью заносятся нужные значения в шейдерные переменные.

```
pWVPVariable->SetMatrix( (float*)&wvp );
pWorldInverseVariable->SetMatrix( (float*)&wi );
pDiffuseVariable->SetResource(pTexture);
pLightVariable->SetFloatVector( (float*)&vLight );
```

На каждый тип шейдерной переменной свой интерфейс и у каждого интерфейса свой набор методов. Интерфейсов много, порядка 12 штук, и именуются они обычно *ID3D10Effect*Variable*, где вместо звездочки вписывается определенный тип переменной: *Matrix*, *Vector*, *Scalar* и т.п. В приведенном выше коде устанавливаются две матрицы, один текстурный буфер и один вектор – позиция источника света.

Настройка массивов

Вывод 3D-модели состоит из трех этапов: установка массивов с геометрией, описывающей модель, настройка параметров материала, при помощи которого будет рисоваться модель и затем команда на начало отрисовки заданными примитивами. Код установки массивов выглядит так:

```
stride = sizeof(VertexT2N3V3);
pDevice->IASetVertexBuffers(0, 1, &pVertexBuffer, &stride, &offset);
pDevice->IASetIndexBuffer(pIndexBuffer, DXGI_FORMAT_R32_UINT, 0);
pDevice->IASetInputLayout(pInputLayout);
```

Так как все геометрические данные (нормаль, позиция и текстурная координата) нашей модели лежат в одном массиве, то драйверу необходимо знать смещение, необходимое для перехода на следующий элемент массива, оно занесено в переменную *stride*.

Наша модель состоит из двух массивов: вершинного и индексного. Вершинный буфер устанавливается методом *IASetVertexBuffers*. Он принимает следующие параметры:

1. *StartSlot* – входной слот, к которому привязывается данный вершинный массив и все последующие (максимальное число слотов может варьироваться от 16 до 32, по умолчанию всегда привязываем к нулевому слоту).
2. *NumBuffers* – число вершинных буферов в массиве, не может превышать число входных слотов, каждый вершинный буфер в массиве привязывается к последующему входному слоту. В нашем случае используется только один вершинный массив.
3. *ppVertexBuffers* – массив вершинных буферов.
4. *pStrides* – массив смещений для каждого вершинного буфера. В нашем случае массив состоит из одного элемента *stride*.
5. *pOffsets* – массив смещений до первого элемента в каждом вершинном буфере.

Установка эффекта и отрисовка геометрии

Финальный этап отрисовки геометрии – установка эффекта и запуск геометрии на растеризацию.

```
pTechnique->GetPassByName("p0")->Apply(0);
pDevice->IASetPrimitiveTopology(
            D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
pDevice->DrawIndexed(meshCount, 0, 0);
```

В прошлых главах был получен интерфейс эффекта, у которого взят указатель на технику. Так как техника может состоять из нескольких проходов, то используется метод *GetPassByName,* возвращающий указатель на интерфейс *ID3D10EffectPass*. Метод *Apply* делает данный проход (все его шейдеры и настройки) текущим.

Затем при помощи метода *IASetPrimitiveTopology* устанавливается информация о том, из каких примитивов будет состоять 3D-модель. В нашем случае это обычные треугольники. Все возможные типы геометрии описаны в справке к *D3D10_PRIMITIVE_TOPOLOGY*.

Весь код, который был ранее, – только подготовка и настройка устройства вывода. Запуск на растеризацию отдает метод *DrawIndexed*. Он принимает три параметра:

- *IndexCount* – число индексов на отрисовку;
- *StartIndexLocation* – номер первого индекса, с которого начинаем рисовать геометрию;
- *BaseVertexLocation* – смещение в байтах от начала вершинного буфера для первой вершины.

В нашем случае достаточно просто указать, из скольки индексов состоит куб.

Вывод результата на экран

Кадр полностью готов, осталось только показать его на экране. Этим занимается интерфейс *IDXGISwapChain*.

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

Первый параметр метода Present отвечает за синхронизацию выводимого кадра и вертикальную развертку, второй – за дополнительные опции вывода кадра (см. справку по *DXGI_PRESENT*).

ЗАКЛЮЧЕНИЕ

На базе данного учебно-методического пособия и методического пособия [2] можно самостоятельно начинать изучение программирования HLSL-шейдеров.

В следующем учебно-методическом пособии будет детально рассмотрен язык программирования шейдеров HLSL, а также примеры создания DirectX 10-приложений. Вместе с этим будет добавлено несколько уроков по созданию и редактированию «десятых» шейдеров в FX Composer.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Мальцев, Д. А. Мультимедиатехнологии. Введение в 3D-программирование / Д. А. Мальцев. – Ульяновск : УлГУ, 2009.
2. Мальцев, Д. А. Мультимедиатехнологии. Основы работы с редактором шейдеров FX Composer / Д. А. Мальцев. – Ульяновск : УлГУ, 2011.
3. Microsoft Developer Network Library, 2012. – URL: http://msdn.microsoft.com/library/default.aspx
4. Microsoft Direct3D 10 Graphics Help, 2012. – URL: http://msdn.microsoft.com/en-us/library/windows/desktop/bb205066(v=vs.85).aspx
5. Джонсон М., Харт. Системное программирование в среде Windows / Джонсон М. Харт. – М. : Вильямс, 2005 – 592 с.
6. Роджерсон, Дейл. Основы COM / Роджерсон Дейл. – М. : Русская редакция, 2000. – 400 с.

Приложение 1

Код для создания 3D-модели кубика

В этом приложении приведен код, необходимый для создания 3D-модели кубика. Каждая вершина состоит из трех элементов: текстурная координата, нормаль и позиция. Куб рисуется при помощи треугольников, на каждую грань куба приходится по два треугольника. Вершины дублируются, так как у них должны быть разные нормали при одинаковой позиции.

```
struct VertexT2N3V3
{
      D3DXVECTOR2       tex;
      D3DXVECTOR3       norm;
      D3DXVECTOR3       pos;
};

#define ARRAY_SIZE(p) (sizeof(p) / sizeof(p[0]))

// вершины меша (кубик), каждые четыре вершины образуют одну грань кубика
VertexT2N3V3 meshV[] =
{
    {
      D3DXVECTOR2(0.0f, 0.0f),
      D3DXVECTOR3(0.0f, 1.0f, 0.0f),
      D3DXVECTOR3(-1.0f,  1.0f, -1.0f)
    },
    {
      D3DXVECTOR2(1.0f, 0.0f),
      D3DXVECTOR3(0.0f, 1.0f, 0.0f),
      D3DXVECTOR3( 1.0f,  1.0f, -1.0f)
    },
    {
      D3DXVECTOR2(1.0f, 1.0f),
      D3DXVECTOR3(0.0f, 1.0f, 0.0f),
      D3DXVECTOR3( 1.0f,  1.0f,  1.0f)
    },
    {
      D3DXVECTOR2(0.0f, 1.0f),
      D3DXVECTOR3(0.0f, 1.0f, 0.0f),
      D3DXVECTOR3(-1.0f,  1.0f,  1.0f)
    },

    {
      D3DXVECTOR2(0.0f, 0.0f),
      D3DXVECTOR3(0.0f, -1.0f, 0.0),
      D3DXVECTOR3(-1.0f, -1.0f, -1.0f)
    },
    {
      D3DXVECTOR2(1.0f, 0.0f),
      D3DXVECTOR3(0.0f, -1.0f, 0.0),
      D3DXVECTOR3( 1.0f, -1.0f, -1.0f)
    },
    {
```

```
  D3DXVECTOR2(1.0f, 1.0f),
  D3DXVECTOR3(0.0f, -1.0f, 0.0),
  D3DXVECTOR3( 1.0f, -1.0f,  1.0f)
},
{
  D3DXVECTOR2(0.0f, 1.0f),
  D3DXVECTOR3(0.0f, -1.0f, 0.0),
  D3DXVECTOR3(-1.0f, -1.0f,  1.0f)
},

{
  D3DXVECTOR2(0.0f, 0.0f),
  D3DXVECTOR3(-1.0f, 0.0f, 0.0),
  D3DXVECTOR3(-1.0f, -1.0f,  1.0f)
},
{
  D3DXVECTOR2(1.0f, 0.0f),
  D3DXVECTOR3(-1.0f, 0.0f, 0.0),
  D3DXVECTOR3(-1.0f, -1.0f, -1.0f)
},
{
  D3DXVECTOR2(1.0f, 1.0f),
  D3DXVECTOR3(-1.0f, 0.0f, 0.0),
  D3DXVECTOR3(-1.0f,  1.0f, -1.0f)
},
{
  D3DXVECTOR2(0.0f, 1.0f),
  D3DXVECTOR3(-1.0f, 0.0f, 0.0),
  D3DXVECTOR3(-1.0f,  1.0f,  1.0f)
},

{
  D3DXVECTOR2(0.0f, 0.0f),
  D3DXVECTOR3(1.0f, 0.0f, 0.0f),
  D3DXVECTOR3( 1.0f, -1.0f,  1.0f)
},
{
  D3DXVECTOR2(1.0f, 0.0f),
  D3DXVECTOR3(1.0f, 0.0f, 0.0f),
  D3DXVECTOR3( 1.0f, -1.0f, -1.0f)
},
{
  D3DXVECTOR2(1.0f, 1.0f),
  D3DXVECTOR3(1.0f, 0.0f, 0.0f),
  D3DXVECTOR3( 1.0f,  1.0f, -1.0f)
},
{
  D3DXVECTOR2(0.0f, 1.0f),
  D3DXVECTOR3(1.0f, 0.0f, 0.0f),
  D3DXVECTOR3( 1.0f,  1.0f,  1.0f)
},

{
  D3DXVECTOR2(0.0f, 0.0f),
  D3DXVECTOR3(0.0f, 0.0f, -1.0),
  D3DXVECTOR3(-1.0f, -1.0f, -1.0f)
},
{
  D3DXVECTOR2(1.0f, 0.0f),
  D3DXVECTOR3(0.0f, 0.0f, -1.0),
```

```
      D3DXVECTOR3( 1.0f, -1.0f, -1.0f)
    },
    {
      D3DXVECTOR2(1.0f, 1.0f),
      D3DXVECTOR3(0.0f, 0.0f, -1.0),
      D3DXVECTOR3( 1.0f,  1.0f, -1.0f)
    },
    {
      D3DXVECTOR2(0.0f, 1.0f),
      D3DXVECTOR3(0.0f, 0.0f, -1.0),
      D3DXVECTOR3(-1.0f,  1.0f, -1.0f)
    },

    {
      D3DXVECTOR2(0.0f, 0.0f),
      D3DXVECTOR3(0.0f, 0.0f, 1.0f),
      D3DXVECTOR3(-1.0f, -1.0f,  1.0f)
    },
    {
      D3DXVECTOR2(1.0f, 0.0f),
      D3DXVECTOR3(0.0f, 0.0f, 1.0f),
      D3DXVECTOR3( 1.0f, -1.0f,  1.0f)
    },
    {
      D3DXVECTOR2(1.0f, 1.0f),
      D3DXVECTOR3(0.0f, 0.0f, 1.0f),
      D3DXVECTOR3( 1.0f,  1.0f,  1.0f)
    },
    {
      D3DXVECTOR2(0.0f, 1.0f),
      D3DXVECTOR3(0.0f, 0.0f, 1.0f),
      D3DXVECTOR3(-1.0f,  1.0f,  1.0f)
    },
};

// индексы меша (кубик), каждые шесть индексов образуют одну грань
// (по два треугольника на каждую)
DWORD meshI[] =
{
    3,1,0,
    2,1,3,

    6,4,5,
    7,4,6,

    11,9,8,
    10,9,11,

    14,12,13,
    15,12,14,

    19,17,16,
    18,17,19,

    22,20,21,
    23,20,22
};

// количество индексов в меше (кубике)
UINT meshCount = ARRAY_SIZE(meshI);
```

Приложение 2

Исходный код HLSL-шейдера из примера

Исходный код HLSL-шейдера из примера реализует простейший вариант диффузного освещения. Также производится перемещение координат объекта в усеченные координаты и наложение текстуры на модель.

```
// Structures

struct VS_INPUT
{
  float2 tex : TEXCOORD;
  float3 norm : NORMAL;
  float4 pos : POSITION;
};

struct PS_INPUT
{
  float4 pos : SV_Position;
  float2 tex : TEXCOORD0;
  float4 color : COLOR0;
};

// Data

matrix wvp;
matrix wi;

Texture2D texDiffuse;
SamplerState samLinear
{
  Filter = MIN_MAG_MIP_LINEAR;
  AddressU = Wrap;
  AddressV = Wrap;
};

float4 light = {0, 0, 0, 1};

// Vertex Shader

PS_INPUT mainVS(VS_INPUT input)
{
  PS_INPUT output = (PS_INPUT)0;
  // texture
  output.tex = input.tex;
  // lighting
  float4 lo = mul(light, wi);
  float4 ln = normalize(lo - input.pos);
  output.color = dot(ln.xyz, input.norm);
  // position
  output.pos = mul(input.pos, wvp);
  return output;
}
```

```
// Pixel Shader

float4 mainPS( PS_INPUT input ) : SV_Target
{
  return texDiffuse.Sample(samLinear, input.tex) * input.color;
}

technique10 tech0
{
  pass p0
  {
    SetVertexShader( CompileShader( vs_4_0, mainVS() ) );
    SetGeometryShader( NULL );
    SetPixelShader( CompileShader( ps_4_0, mainPS() ) );
  }
}
```