Game Development with DirectX 9

Contents

Introduction

The goal of this document gives an introduction to game programming with DirectX 9. This will be done by explaining the basics of each topic/technique and giving example code. At the end, I will show a sample application called Project with source code that uses the explained techniques. This main sample will be discussed in the final chapter. It can be downloaded from: http://www.cs.vu.nl/~tljchung/directx/samples/Project.zip.
Most topics discussed are based on Frank D. Luna's book, "Introduction to 3D GAME Programming with DirectX 9.0". The book itself is not necessary when reading this document.

Prerequisites

This document designed to be an introduction to DirectX. No knowledge of DirectX is required. However, some prerequisites are.
The first one is mathematical knowledge. The reader is assumed to be familiar with terms like vectors, normals, matrices and transformation.
Second, knowledge of C/C++ and data structures like arrays and lists is also required. Having knowledge of Windows programming is helpful, but not a must.

Required tools/downloads

The required development tools are Visual C++ 6.0 or 7.0 (.NET), DirectX 9.0c End-User Runtime and the DirectX 9.0c SDK. Section 1.4.4 explains how to compile DirectX programs with Visual C++ 6. View the following PDF document for Visual Studio .NET 2002: http://www.moon-labs.com/resources/sample_setup.pdf .
The SDK version used in this document is DirectX 9.0 SDK Update - (June 2005). It can be downloaded from: http://msdn.microsoft.com/directx/.
The runtime will be installed when you install the SDK. To view information regarding your DirectX installation, select Run from the Start menu in Windows. Enter *dxdiag* and click OK.
The official DirectX documentation can be found at the windows start menu. By default at: "Microsoft DirectX 9.0 SDK Update (June 2005)". The documentation works very well as a reference. All DirectX functions, objects etc. can be found there. It also contains some tutorials.
This document uses the examples given in Luna's book. As said before, the book itself is not necessary, but I do recommend downloading the example programs from this book. Visit http://www.moon-labs.com/ml_book_samples.htm or http://www.wordware.com/files/dx9/ for downloading the examples.
For more help, you could also visit the handy forum found at the moon-labs link.

Conventions

Pieces of code will be written in `Courier font`. Comments in code have a `green color`. Large blocks of code will have a grey background.

```
// comments
int x = 1;
int y = 25;
```

D3DX Library

The D3DX library is shipped with the DirectX SDK and contains features that would be a chore for the programmer to implement. Like math, texture and image operations.
The library is used in this document, because D3DX:

- contains handy functions, for example loading an image file. This way, we can avoid spending pages explaining how to load images.
- is also used other developers
- is fast.
- has been thoroughly tested.

Introduction to DirectX 9

The DirectX SDK is an API developed by Microsoft that allows programmers to access the hardware directly in Windows. DirectX consists of several components, namely:

- DirectX Graphics: combines the DirectDraw and Direct3D components of previous DirectX versions into a single application programming interface (API) that you can use for all graphics programming. The component includes the Direct3D extensions (D3DX) utility library, which simplifies many graphics programming tasks.
- DirectSound: can be used in the development of high-performance audio applications that play and capture waveform audio.
- DirectMusic: provides a complete solution for both musical and non-musical soundtracks based on waveforms, MIDI sounds, or dynamic content authored in DirectMusic Producer.
- DirectInput: provides support for a variety of input devices, including full support for force-feedback technology.
- DirectPlay: provides support for multiplayer networked games.

This document will only discuss Direct3D.

Direct3D

1. Direct3D Initialization

The Direct3D initialization process assumes that the programmer is familiar with basic graphics concepts and some fundamental Direct3D types. These requirements will be addressed in this chapter.

Keywords: HAL, REF Device, `D3DDEVTYPE`, COM, surface, multisampling, memory pools, swap chain, depth buffer, device capabilities, `D3DCAPS9`, `IDirect3D9`, `IDirect3DDevice9`, `D3DPRESENT_PARAMETERS`

1.1 Overview

Direct3D is a low-level graphics API that enables us to render 3D worlds using 3D hardware acceleration. It can be thought of as a mediator between the application and the graphics device (3D hardware).

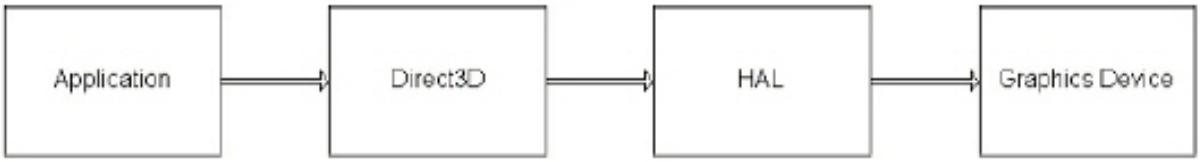


Figure 1.1: The relationship between the application, Direct3D, and the hardware

The Direct3D part of figure 1.1 represents the set of features which are provided by Direct3D. However, it doesn't automatically mean that the hardware supports it.

HAL (Hardware Abstraction Layer) is the set of device-specific code that instructs the device to perform an operation and is implemented by the manufacturer. Direct3D uses HAL to

interact with the graphics device. This way it doesn't need to know about the specific details of a device. When Direct3D calls a function that is not implemented by the HAL, it results in failure.

1.1.1 REF Device

To test a function that is not implemented by your device, you can use the REF device (reference rasterizer). For example, you could test vertex and pixel shaders with the REF device, even if your device does not support it. However, the REF device is slow. It should only be used for development and not to be distributed to end users. That is why it is only shipped with the SDK.

1.1.2 D3DDEVTYPE

In code, a HAL device is specified by `D3DDEVTYPE_HAL`, which is a member of the `D3DDEVTYPE` enumerated type. Similarly, a REF device is specified by `D3DDEVTYPE_REF`, which is also a member of the `D3DDEVTYPE` enumerated type. We will be asked to specify which type to use when creating our device.

1.1.3 COM

Component Object Model (COM) is the technology that allows DirectX to be language independent and have backward compatibility. We usually refer to a COM object as an interface, which for our purposes can be thought of and used as a C++ class. Most of the details of COM are transparent to us when programming DirectX with C++. The only thing that we must know is that we obtain pointers to COM interfaces through special functions or the methods of another COM interface; we do not create a COM interface with the C++ *new* keyword. In addition, when we are done with an interface, we call its `Release` method rather than `delete` it. COM objects perform their own memory management.
Knowing more details on COM is not necessary for using DirectX effectively.

1.2 Preliminaries

The following sections introduce the basic graphics concepts and Direct3D types.

1.2.1 Surfaces

A surface is a matrix of pixels that Direct3D uses primarily to store 2D image data. See figure 1.2 for some components of a surface.

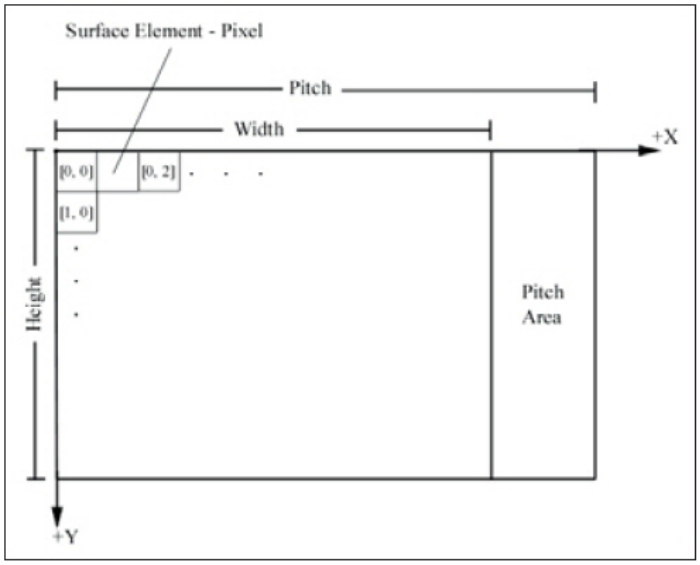


Figure 1.2: A surface

Pixel data is stored in a linear array. The width and height are measured in pixels, while the pitch is measured in bytes. Depending on the hardware, the pitch can be wider than the width. So you cannot assume that pitch = width * sizeof(pixelFormat).
The `IDirect3DSurface9` interface provides methods for reading, writing to a surface as well as retrieving information about the surface.
The most important methods are:

- `LockRect`: This method allows us to obtain a pointer to the surface memory. Then, with some pointer arithmetic, we can read and write to each pixel in the surface.
- `UnlockRect`: After you have called LockRect and are done accessing the surface's memory, you must unlock the surface by calling this method.
- `GetDesc`: This method retrieves a description of the surface by filling out a `D3DSURFACE_DESC` structure.

The following code shows how to lock a surface and color each pixel red.

```
// Assume  surface is a pointer to an IDirect3DSurface9 interface.
// Assumes a 32-bit pixel format for each pixel.

// Get the surface description.
D3DSURFACE DESC surfaceDesc;
_surface->GetDesc(&surfaceDesc);

// Get a pointer to the surface pixel data
D3DLOCKED_RECT lockedRect;
 surface->LockRect(
      &lockedRect,     // pointer to receive locked data
      0,               // lock entire surface
      0);              // no lock flags specified

// Iterate through each pixel in the surface and set it to red.
DWORD* imageData = (DWORD*)lockedRect.pBits;
for(int i = 0; i < surfaceDesc.Height; i++) {
      for(int j = 0; j < surfaceDesc.Width; j++) {
      // index into texture, note we use the pitch and divide
      // by four since the pitch is given in bytes and there
      // are 4 bytes per DWORD.
            int index = i * lockedRect.Pitch / 4 + j;
            imageData[index] = 0xffff0000; // red
      }
}

_surface->UnlockRect();
```

1.2.2 Multisampling

Multisampling is a technique used to make blocky-looking images that can result when representing images as a matrix of pixels look smoother
The `D3DMULTISAMPLE_TYPE` enumerated type consists of values that allow us to specify the level of multisampling of a surface. They are:

- `D3DMULTISAMPLE_NONE`: Specifies no multisampling
- `D3DMULTISAMPLE_1_SAMPLE ... D3DMULTISAMPLE_16_SAMPLE`: Specifies multisampling levels from 1 to 16

Multisampling is not used here, because it slows down the application to much. If you wish to include it, use the `IDirect3D9::CheckDeviceMultiSampleType` method to verify that

the graphics device supports the multisampling type that you wish to use and check for valid quality levels.

1.2.3 Pixel Formats

We often need to specify the pixel format of Direct3D resources when we create a surface or texture. The format of a pixel is defined by specifying a member of the `D3DFORMAT` enumerated type. Some formats are:

- `D3DFMT_R8G8B8`: Specifies a 24-bit pixel format where, starting from the leftmost bit, 8 bits are allocated for red, 8 bits are allocated for green, and 8 bits are allocated for blue. In other words:
  - 8 bits red
  - 8 bits green
  - 8 bit blue
- `D3DFMT_X8R8G8B8`: 32-bit pixel format
  - 8 bits not used
  - 8 bits red
  - 8 bits green
  - 8 bits blue
- `D3DFMT_A8R8G8B8`: 32-bit pixel format
  - 8 bits alpha
  - 8 bits red
  - 8 bits green
  - 8 bits blue
- `D3DFMT_A16B16G16R16F`: 64-bit, floating-point pixel format
  - 16 bits alpha
  - 16 bits blue
  - 16 bits green
  - 16 bits red
- `D3DFMT_A32B32G32R32F`: 128-bit, floating-point pixel format.
  - 32 bits alpha
  - 32 bits blue
  - 32 bits green
  - 32 bits red

Look up `D3DFORMAT` in the SDK documentation for a complete list of all the supported pixel formats.

1.2.4 Memory Pools

Surfaces and other Direct3D resources can be placed in a variety of memory pools. The memory pools available are specified in the `D3DPOOL` enumerated type:

- `D3DPOOL_DEFAULT`: The default memory pool instructs Direct3D to place the resource in the memory that is best suited for the resource type and its usage. This may be video memory, AGP memory, or system memory. Note that resources in the default pool must be destroyed (released) prior to an `IDirect3DDevice9::Reset` call, and must be reinitialized after the reset call.
- `D3DPOOL_MANAGED`: Resources placed in the manage pool are managed by Direct3D (that is, they are moved to video or AGP memory as needed by the device automatically). In addition, a back-up copy of the resource is maintained in system memory. When resources are accessed and changed by the application, they work with

the system copy. Then, Direct3D automatically updates them to video memory as needed.

- `D3DPOOL_SYSTEMMEM`: Specifies that the resource be placed in system memory
- `D3DPOOL_SCRATCH`: Specifies that the resource be placed in system memory. The difference between this pool and `D3DPOOL_ SYSTEMMEM` is that these resources must not follow the graphics device's restrictions. Consequently, the device cannot access resources in this pool. But the resources can be copied to and from each other.

1.2.5 Swap Chain

Direct3D maintains usually two or three surfaces, called a swap chain that is represented by the IDirect3DSwapChain9 interface. Direct3D manages it and we rarely need to manipulate it. Instead, the purpose of it will be outlined.
Basically, there is a *front buffer* that corresponds to the image displayed by the monitor. If the application renders frames faster than the monitor's refresh rate, the application will render to an off-screen surface called *back buffer* first. The back buffer will be promoted to be the front buffer when the monitor has finished displaying the original front buffer. This process is called *presenting*.

1.2.6 Depth Buffers

In order for Direct3D to determine which pixels of an object are in front of another, it uses a technique called *depth buffering* or *z-buffering*. This technique compares the depth each pixel competing for a particular pixel location. The pixel closest to the camera is written.
The higher the depth buffer format, the more accurate it is. Most applications work fine with a 24-bit depth buffer.

- `D3DFMT_D32`: Specifies a 32-bit depth buffer
- `D3DFMT_D24S8`: Specifies a 24-bit depth buffer with 8 bits reserved as the stencil buffer[i]
- `D3DFMT_D24X8`: Specifies a 24-bit depth buffer only
- `D3DFMT_D24X4S4`: Specifies a 24-bit buffer with 4 bits reserved for the stencil buffer
- `D3DFMT_D16`: Specifies a 16-bit depth buffer only.

1.2.7 Vertex Processing

Vertices are the building blocks for 3D geometry, and they can be processed either in software (software vertex processing) or in hardware (hardware vertex processing). Software vertex processing is always supported and can always be used. Hardware vertex processing can only be used if the graphics card supports vertex processing in hardware.
Hardware vertex processing is always preferred since dedicated hardware is faster than software. Furthermore, performing vertex processing in hardware unloads calculations from the CPU.

1.2.8 Device Capabilities

We can check if a device supports a feature by checking the corresponding data member or bit in the `D3DCAPS9` structure.
Suppose we wish to check if a hardware device is capable of doing vertex processing in hardware (or in other words, whether the device supports transformation and lighting calculations in hardware). By looking up the `D3DCAPS9` structure in the SDK documentation, we find that the bit `D3DDEVCAPS_HWTRANSFORMANDLIGHT` in the data member

[i] The stencil buffer is a more advanced topic and is explained in chapter 9

`D3DCAPS9::DevCaps` (device capabilities) indicates whether the device supports transformation and lighting calculations in hardware. Our test then, assuming caps is a `D3DCAPS9` instance and has already been initialized, is:

```
bool supportsHardwareVertexProcessing;
// If the bit is "on" then that implies the hardware device
// supports it.
if( caps.DevCaps & D3DDEVCAPS HWTRANSFORMANDLIGHT ) {
      // Yes, the bit is on, so it is supported.
      supportsHardwareVertexProcessing = true;
} else {
      // No, the bit is off, so it is not supported.
      hardwareSupportsVertexProcessing = false;
}
```

1.3 Initializing Direct3D

To initialize Direct3D, these 4 steps should be taken.

1.3.1 Acquire an IDirect3D9 interface

The IDirect3D9 interface is used for finding out information about the physical hardware devices on a system and creating the `IDirect3DDevice9` interface, which is our C++ object that represents the physical hardware device we use for displaying 3D graphics.

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

To guarantee the application is built with the correct header files, the parameter `D3D_SDK_VERSION` should be used.

1.3.2 Check the device capabilities

With the `D3DCAPS9` structure, we can see if the primary display adapter (primary graphics card) supports hardware vertex processing or not. We need to know if it can in order to create the `IDirect3DDevice9` interface.

```
HRESULT GetDeviceCaps(
      // the physical display adapter that we are
      // going to get the capabilities of
      UINT Adapter,
      //hardware or software device
      D3DDEVTYPE DeviceType,
      //results are stored in this structure
      D3DCAPS *pCaps);
```

We can find if the hardware supports hardware vertex processing with the following if-statement (as seen in 1.2.8).

```
if (pCaps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT)
```

1.3.3 Initialize an instance of the D3DPRESENT_PARAMETERS structure

This structure consists of a number of data members that allow us to specify the characteristics of the `IDirect3DDevice9` interface that we are going to create.
It is defined as:

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

See the SDK documentation for a description of this structure.

1.3.4 Create the IDirect3DDevice9 object

This can be done with the `CreateDevice()` method. The `D3DPRESENT_PARAMETERS` structure from the previous step is needed.

```
HRESULT CreateDevice(
      // The physical display adapter
      UINT Adapter,
      // Hardware or software device
      D3DDEVTYPE DeviceType,
      // Handle the window that the device will be associated with
      // usually the window the device will draw onto
      HWND hFocusWindow,
      // Combination of one or more options that control device
      // creation. See D3DCREATE.
      DWORD BehaviorFlags,
      // Defines some of the characteristics of the device
      D3DPRESENT_PARAMETERS *pPresentationParameters,
      // Returns the created device
      IDirect3DDevice9 **ppReturnedDeviceInterface
);
```

1.4 Example Program, D3D Init

The first example, called D3D Init, creates and initializes a Direct3D application and clears the screen to black. As mentioned, the example programs can be downloaded from http://www.moon-labs.com/ml_book_samples.htm.

1.4.1 Framework

The first example program from Luna's book was chosen, because it provides us with a nice framework.
The framework contains the files d3dUtility.cpp, d3dUtility.h and a main program file. Functions for handling the window and Direct3D initialization are implemented in d3dUtility.cpp. The main file will contain three specific functions:

- `bool Setup()`: Set up anything needed for the example, such as allocating resources and checking device capabilities.
- `void Cleanup()`: Frees anything allocated in the Setup() function.

- `bool Display(float timeDelta)`: Contains all of the drawing code and code which occur on a frame-by-frame basis, such as updating object positions. The `timeDelta` parameter is the time elapsed between each frame.

All examples given in this document will fill out these three functions.

1.4.2 d3dUtility.h/cpp

Let's take a look at the functions provided by d3dUtility.h/cpp.

```
// Include the main Direct3DX header file. This will include the
// other Direct3D header files we need.
#include <d3dx9.h>

namespace d3d {
      bool InitD3D(
            HINSTANCE hInstance, // [in] Application instance.
            int width, int height, // [in] Back buffer dimensions.
            bool windowed,  // [in] Windowed (true)or
                            //full screen (false).
            D3DDEVTYPE deviceType, // [in] HAL or REF
            IDirect3DDevice9** device); // [out] The created device.

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

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

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

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

- `InitD3D`: Initializes a main application window and implements the Direct3D initialization code discussed in chapter 1.3. It outputs a pointer to a created `IDirect3DDevice9` interface if the function returns successfully. The parameters allow us to specify the window's dimensions and whether it should run in windowed mode or full-screen mode.
- `EnterMsgLoop`: This function wraps the application message loop. The parameter *ptr_display is a pointer to the display function, which implements the sample's drawing code. The message loop function needs to know the display function so that it can call it and display the scene during idle processing. It also calculates the time between frames.

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

      static float lastTime = (float)timeGetTime();

      while(msg.message != WM QUIT) {
            if(::PeekMessage(&msg, 0, 0, 0, PM REMOVE)) {
                  ::TranslateMessage(&msg);
                  ::DispatchMessage(&msg);
            } else {
                  float currTime  = (float)timeGetTime();
                  float timeDelta = (currTime - lastTime)*0.001f;

                  ptr_display(timeDelta);

                  lastTime = currTime;
            }
      }
      return msg.wParam;
}
```

- `Release`: This template function is designed as a convenience function to release COM interfaces and set them to null.
- `Delete`: This template function is designed as a convenience function to delete an object on the free store and set the pointer to null.
- `WndProc`: The window procedure declaration for the main application window

1.4.3 D3D Init Implementation

We can finally describe the main file of this sample program, d3dInit.cpp. It starts with including d3dUtility.h and instantiating a global variable for the device:

```
#include "d3dUtility.h”

IDirect3DDevice9* Device = 0;
```

The framework functions `Setup()` and `Cleanup()` are empty, because we have nothing to setup in this sample.

Display

The `Display()` method calls `IDirect3DDevice9::Clear` to clear the back buffer to black and the depth buffer to 1.0.

```
bool Display(float timeDelta) {
      if( Device ) {
            Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                        0x00000000, 1.0f, 0);

            // Swap the back and front buffers.
            Device->Present(0, 0, 0, 0);
      }
      return true;
}
```

IDirect3DDevice9:Clear

The `IDirect3DDevice9::Clear` method is shown below. For more information, see the SDK documentation.

```
HRESULT Clear(
      //Number of rectangles in the array at pRects
      DWORD Count,
      //Array of screen rectangles to clear
      //allows clearing part of the surface
      const D3DRECT *pRects,
      //Which surface to clear
      DWORD Flags,
      //Clear a render target to this color
      D3DCOLOR Color,
      //Clear the depth buffer to this value, ranges from 0 to 1.
      float Z,
      //The value we wish to set the stencil buffer to
      DWORD Stencil
);
```

The value 0x00000000 is used for representing the color black. We could use the macro `D3DCOLOR_XRGB(r,g,b)` to get this value. All values are to be in the range 0 to 255. For black, use `D3DCOLOR_XRGB(0,0,0)`. See chapter 4, Color.

WndProc

The window procedure method allows us to exit the application with the Escape key.

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

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

WinMain

The execution begins at the `WinMain` function. It performs the following steps:

1. Initializes the main display window and Direct3D
2. Calls the Setup routine to set up the application
3. Enters the message loop using Display as the display function
4. Cleans up the application and finally releases the IDirect3DDevice9 object

```
int WINAPI WinMain(HINSTANCE hinstance,
                           HINSTANCE prevInstance,
                           PSTR cmdLine,
                           int showCmd)
{
      if(!d3d::InitD3D(hinstance,
            640, 480, true, D3DDEVTYPE HAL, &Device))
      {
```

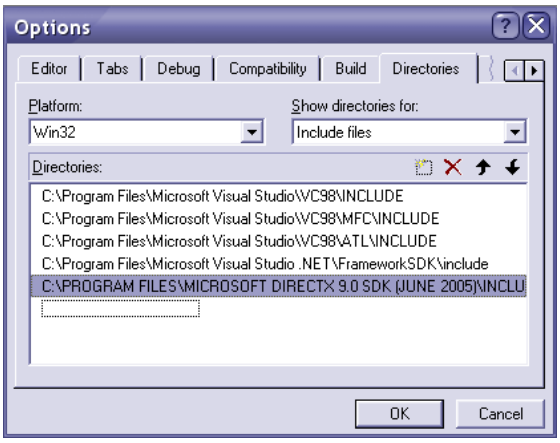
```
            ::MessageBox(0, "InitD3D() - FAILED", 0, 0);
            return 0;
      }

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

      d3d::EnterMsgLoop( Display );
      Cleanup();
      Device->Release();

      return 0;
}
```

1.4.4 Compiling with Visual C++ 6.0

To compile the example program with Microsoft Visual C++ 6.0 (VC++6), certain steps need to be taken. The D3D Init example contains the three files d3dUtility.cpp, d3dUtility.h and the main file d3dInit.cpp. There is no VC++6 project file provided, we will have to make it ourselves.

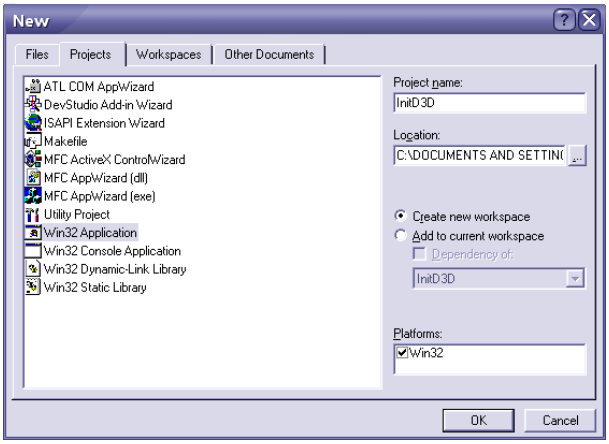
Settings

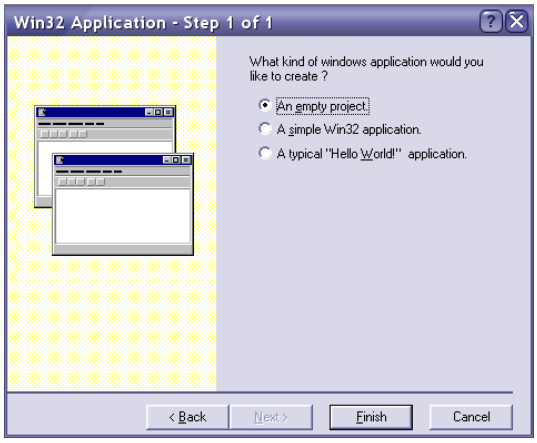
Before starting a new DirectX9 project with VC++6, make sure your Include and Library paths are set correctly. In the menu: **Tools->Options, Directories tab**. If your DirectX9 Include path is not there yet, make sure "Include files" is selected from the dropdown menu and click the "new" button. Select the right directory located at: "<DirectX9 path>/Include". Next, select "Library files" from the dropdown menu, click "new" again and add: "<DirectX9 path>/Lib/x64" or "<DirectX9 path>/Lib/x86", depending on your PC.



New Project

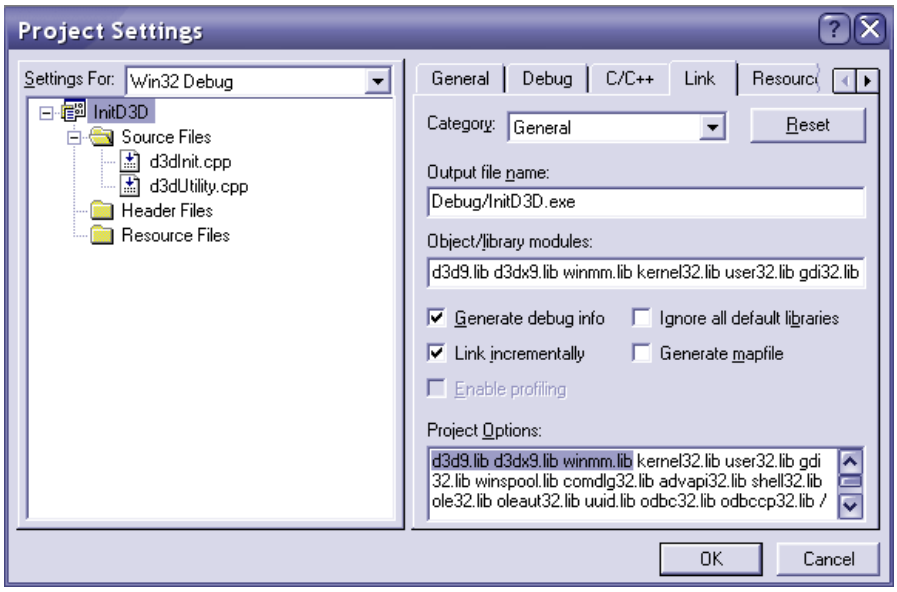
In VC++6 select **File->New**. Select the "Projects" tab and choose "Win32 Application". Enter a project name, for example "InitD3D" and click "OK". In the next screen just leave it to "An empty project" and click "Finish". You will see a confirmation window; just click "OK".





Project Settings

VC++6 needs to know about the DirectX9 libraries. Select **Project->Settings** and click on the project name (InitD3D). Select the "Link" tab, in the field "Project Options" type in "d3d9.lib d3dx9.lib winmm.lib" without the quotes. This is highlighted in the screenshot below. Click "OK" to save.

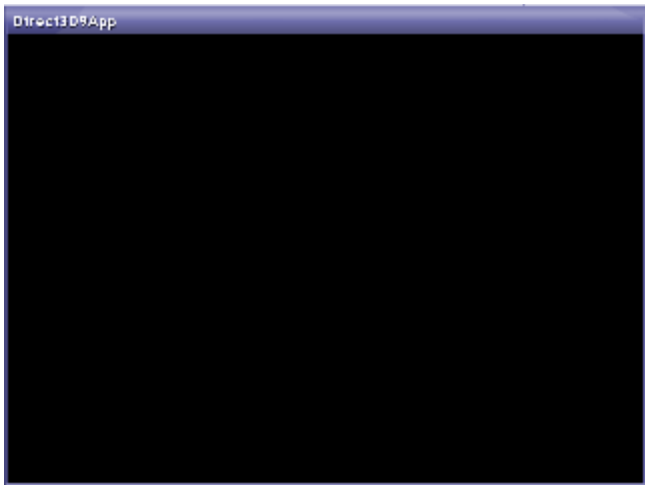


Adding the source files

Select "FileView" on the left bottom of the screen, right click on the "Source Files" map and select "Add Files to Folder…". Select the two cpp files. You can also add the header files to the "Header Files" folder if you want to.

Compiling

We can finally compile the project. Select Build (F7) and after that select Execute Program (Ctrl+F5). You should now see the black application screen.



You just compiled your first DirectX9 program with VC++6! Congratulations!

1.5 The CreateDevice example

The CreateDevice example program comes from the DirectX SDK documentation. It roughly follows the same ideas as D3D Init. It's recommended to take a look at this sample to gain a different perspective.
The sample can easily be obtained via the DirectX Sample Browser. Select the Direct3D option and make sure the boxes C++ and Tutorials are selected. Look for "Tutorial1: CreateDevice" and choose "Install Project" to save it. CreateDevice consists of one source file CreateDevice.cpp and comes with several different versions of Visual Studio .Net projects. If you don't have Visual Studio .Net, the procedure explained in 1.4.4 can be followed to compile CreateDevice with VC++6.

1.5.1 Differences

Some differences between CreateDevice and D3D Init:

- CreateDevice consists of one file; D3D Init puts the window and Direct3D initialization in different files.
- CreateDevice does not check the device capabilities. It simply sets the vertex processing to `D3DCREATE_SOFTWARE_VERTEXPROCESSING`.
- CreateDevice uses `WS_OVERLAPPEDWINDOW` to create the window. D3D Init uses `WS_EX_TOPMOST`.
  An application window with `WS_OVERLAPPEDWINDOW` contains the three window buttons at the top right (minimize, maximize and close).

1.6 Summary

- Direct3D is a mediator between the programmer and the graphics hardware. The programmer calls a Direct3D function, which in turn has the physical hardware perform the operation by interfacing with the device's HAL (Hardware Abstraction Layer).
- The REF device allows developers to test features that Direct3D exposes but are not implemented by available hardware.
- Component Object Model (COM) allows DirectX to be language independent and have backward compatibility. Programmers don't need to know the details of COM and how it works.
- The `IDirect3D9` interface can be used to find out the capabilities of a device. It is also used to create the `IDirect3DDevice9` interface.

- The `IDirect3DDevice9` interface is our C++ object that represents the physical hardware device we use for displaying 3D graphics.
- The examples in this document use the utility files d3dUtility.h/cpp and always fill the methods `Setup`, `Clean` and `Display`.
- Visual C++ 6.0 can be used to compile DirectX applications with the latest DirectX SDK, but needs to be configured manually.

2. The Rendering Pipeline

The rendering pipeline is responsible for creating a 2D image given a geometric description of the 3D world and a virtual camera. The objectives of this chapter:

- Find out how we represent 3D objects in Direct3D.
- Learn how we model the virtual camera.
- Understanding the rendering pipeline.

Keywords: triangle mesh, vertex format, triangle, camera, frustum, clipping, rendering pipeline, local space, world space, view space, backface culling, projection, viewport, `D3DVIEWPORT9`, rasterization

2.1 Model Representation

A scene is a collection of objects or models. An object is represented as triangle meshes, a list of triangles that approximates the shape and contours of the object. Illustrated in figure 2.1.

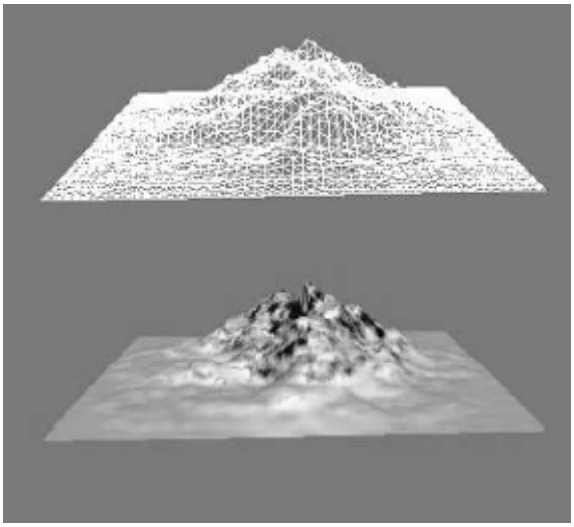


Figure 2.1: Terrain made of triangles

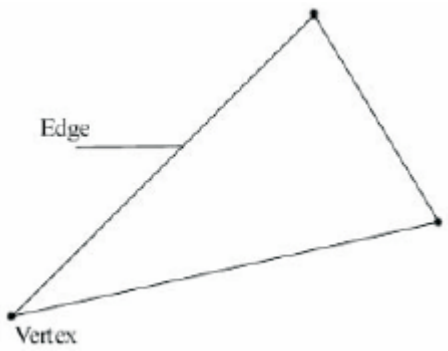


Figure 2.2: Triangle defined by three vertices

Polygons, primitives and mesh geometry are terms used to refer to the triangles of a mesh. The point where two edges on a polygon meet is a vertex. To describe a triangle we specify the three point locations that correspond to the three vertices of the triangle (see Figure 2.2). Then to describe an object, we specify the triangles that make it up.

2.1.1 Vertex Formats

A vertex in Direct3D can consist of additional properties besides special location. For instance a color and normal property. Direct3D allows us to construct our own vertex formats. To create a custom vertex format, we first create a structure. For instance, a custom color vertex:

```
struct ColorVertex
{
    FLOAT x, y, z; // position for the vertex.
    DWORD color;   // The vertex color.
};
```

We need to describe the way the vertices are formatted by using a combination of flexible vertex format (FVF) flags.

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

The flags in the code above say that the vertex structure contains a position property and a diffuse color property. These flags must be in the same order as the specified data in the custom vertex structure.
Look up `D3DFVF` in the documentation for a complete list of the available vertex format flags.

2.1.2 Triangles

To construct an object, we create a triangle list that describes the shape and contours of the object. A triangle list contains the data for each individual triangle that we wish to draw. For example, to construct a rectangle, we break it into two triangles, as seen in Figure 2.3, and specify the vertices of each triangle.

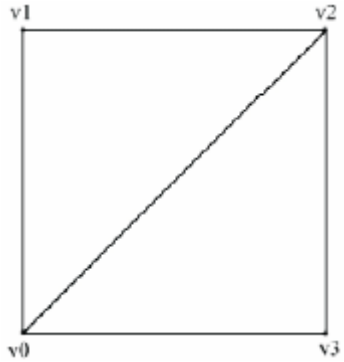


Figure 2.3: A rectangle constructed with two triangles

```
Vertex rect[6] = {v0, v1, v2, // triangle0
v0, v2, v3}; // triangle1
```

2.1.3 Indices

The triangles that form a 3D object often share many of the same vertices. Like the cube shown in figure 2.4. It has eight unique vertices. Many of these vertices would be duplicated to form the triangle list for the cube.

Figure 2.4: A cube made with triangles

The concept of indices is introduced to solve this. It works like this: There is a vertex and index list. The vertex list consists of all the unique vertices; the index list contains values that index into the vertex list to define how they are to be put together to form triangles.

In the rectangle sample, the vertex list and index list would be constructed as follows:

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

WORD indexList[6] = {0, 1, 2, // triangle0
                     0, 2, 3}; // triangle1
```

In other words, `indexList` says that triangle0 is built from elements 0,1 and 2 of the vertex list. Triangle1 is built from 0, 2 and 3.

2.2 The Virtual Camera

The camera specifies what part of the world the viewer can see and thus what part of the world for which we need to generate a 2D image.
The volume of space the camera "sees" is called a *frustum*, defined by the field of view angles and the near and far planes. Objects that are not inside this volume cannot be seen and should be discarded from further processing. This is called *clipping*.

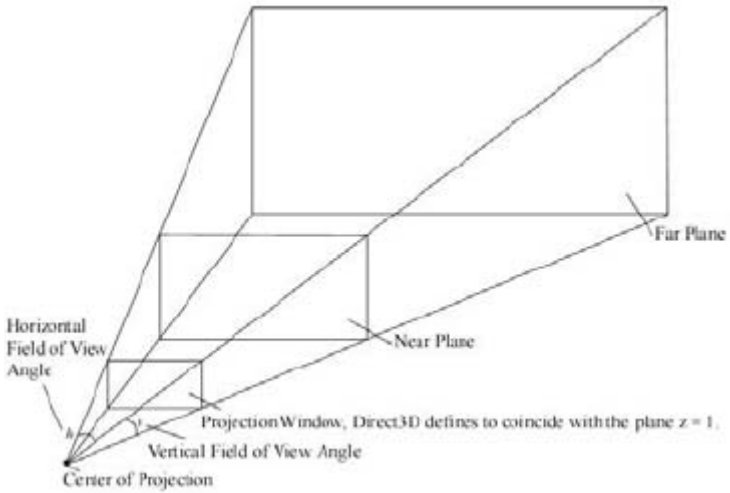


Figure 2.5

The *projection* window is the 2D area that the 3D geometry inside the frustum gets projected onto to create the 2D image representation of the 3D scene. It is important to know that we define the projection window with the dimensions min = (–1, –1) and max = (1, 1).
To simplify some of the drawings that are to follow in this tutorial, we make the near plane and *projection plane* (plane the projection window lies on) coincide. Also, note that Direct3D defines the projection plane to be the plane z = 1.

2.3 The Rendering pipeline

The series of operations to produce a 2D representation of a 3D scene, is called the rendering pipeline. Figure 2.6 shows the stages of the pipeline.

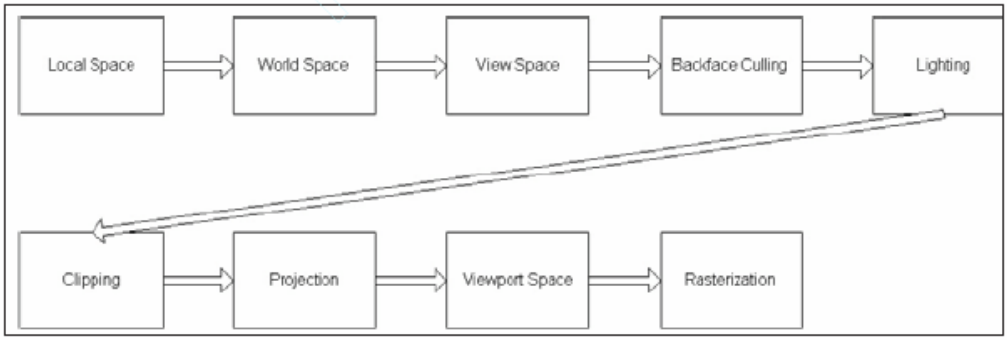


Figure 2.6: The rendering pipeline stages

Several of the stages in the pipeline transform geometry from one coordinate system to another. The transformations are done using matrices. Direct3D is set up to do the transformation calculations for us. All we must do is supply the desired transformation matrix that describes the transformation needed to go from one system to the next. We supply a matrix using the `IDirect3DDevice->SetTransform` method. This method takes a parameter describing the transformation type and a matrix that represents the transformation. To set the transformation needed to go from local space to world space, we would write:

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

2.3.1 Local Space

We define an object's triangle list in the coordinate system called the local space. This simplifies the modeling process, because building a model around its own local coordinate system is easier than building a model directly into the world. For instance, you could construct a model without regard to its position or size.

2.3.2 World Space

The constructed models, each residing in their local space, are brought together to form the scene in one global (world) coordinate system. Transforming objects from local space to world space is called *world transform*. This usually consists of translations, rotations and scaling. The world transformation sets up all the objects in the world in relationship to each other in position, size, and orientation.
The world transformation is represented with a matrix and set with Direct3D using the `IDirect3DDevice9::SetTransform` method with `D3DTS_WORLD` as the transform type. For example, suppose we want to position a cube at the point (–3, 2, 6) in the world and a sphere at the point (5, 0, –2). We would write:

```
// Build the cube world matrix that only consists of a translation.
D3DXMATRIX cubeWorldMatrix;
D3DXMatrixTranslation(&cubeWorldMatrix, -3.0f, 2.0f, 6.0f);

// Build the sphere world matrix that only consists of a
// translation.
D3DXMATRIX sphereWorldMatrix;
D3DXMatrixTranslation(&sphereWorldMatrix, 5.0f, 0.0f, -2.0f);

// Set the cube's transformation
Device->SetTransform(D3DTS_WORLD, &cubeWorldMatrix);
drawCube(); // draw the cube

// Now since the sphere uses a different world transformation, we
// must change the world transformation to the sphere's. If we
// don't change this, the sphere would be drawn using the previously
// set world matrix - the cube's.
Device->SetTransform(D3DTS_WORLD, &sphereWorldMatrix);
drawSphere(); // draw the sphere
```

2.3.3 View Space

The camera is transformed to the origin of the world system to make things easier. It is rotated so that the camera is looking down the positive z-axis. All geometry in the world is transformed along with the camera so that the view of the world remains the same. This transformation is called the *view space transformation*.

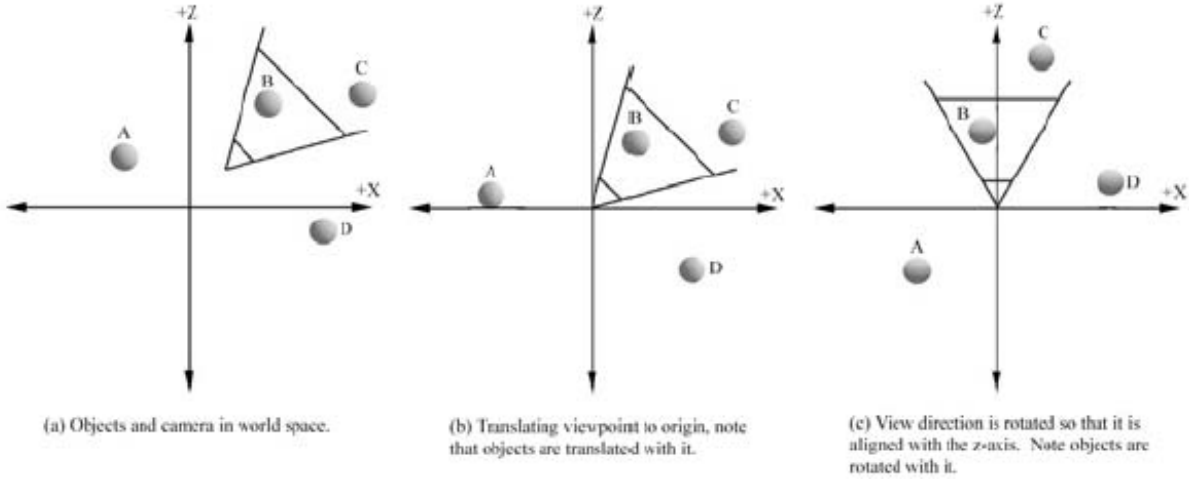


Figure 2.7: Transformation from world space to view space

The view space transformation matrix can be computed using the following D3DX function:

```
D3DXMATRIX *D3DXMatrixLookAtLH(
      D3DXMATRIX* pOut, // pointer to receive resulting view matrix
      CONST D3DXVECTOR3* pEye, // position of camera in world
      CONST D3DXVECTOR3* pAt, // point camera is looking at in world
      CONST D3DXVECTOR3* pUp // the world's up vector - (0, 1, 0)
};
```

Suppose we want to position the camera at the point (5, 3, –10) and have the camera look at the center of the world (0, 0, 0). We can then build the view transformation matrix by writing:

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

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

The view space transformation is set with the `IDirect3DDevice9::SetTransform` method with `D3DTS_VIEW` as the transform type:

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

2.3.4 Backface Culling

A polygon has a front and a back side. In general, the back sides are never seen because of enclosed volumes like boxes, cylinders, tanks etc. The camera should not be allowed to enter the space inside the object.
In figure 2.8a we can see an object where the front sides have an arrow sticking out. A polygon whose front side faces the camera is called a *front facing* polygon, and a polygon whose front side faces away from the camera is called a *back facing* polygon.

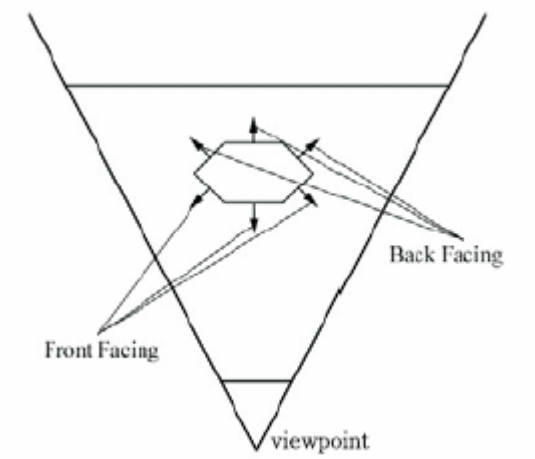


Figure 2.8a

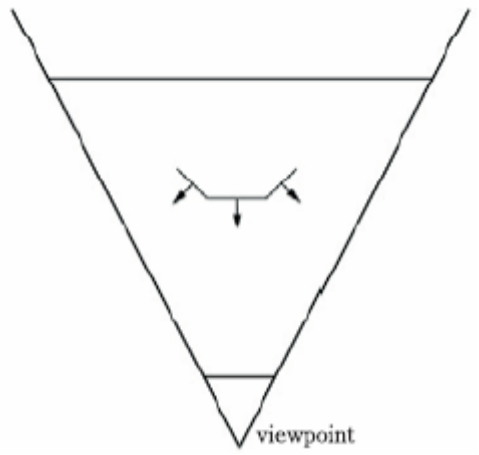


Figure 2.8b: After back face culling

Figure 2.8b shows the same object after *backface culling*. An operation that discards the back facing polygons from further processing. From the camera's viewpoint, the same scene will be drawn because the back faces were obscured anyway and would never have been seen.
By default, Direct3D treats triangles with vertices specified in a clockwise winding order (in view space) as front facing. Triangles with vertices specified in counterclockwise winding orders (in view space) are considered back facing.
The culling behavior can be changed with:

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

Where `Value` can be one of the following:

- `D3DCULL_NONE`: Disables back face culling entirely
- `D3DCULL_CW`: Triangles with a clockwise wind are culled.
- `D3DCULL_CCW`: Triangles with a counterclockwise wind are culled. This is the default state.

2.3.5 Clipping

A triangle can be at one of these three locations with regards to the frustum:

1. Completely inside
2. Completely outside
3. Partially inside

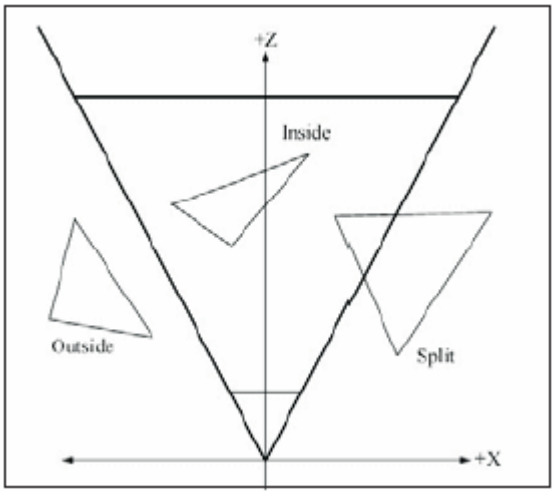


Figure 2.9: Clipping

The part inside the frustum is kept and the part outside is culled. This is called *clipping*.

2.3.6 Projection

The process of going from an *n* dimension to an *n–1* dimension (e.g. from 3D to 2D) is called *projection*. A particular way called *perspective projection* projects geometry in such a way that objects farther away from the camera appear smaller than those near the camera. This type of projection allows us to represent a 3D scene on a 2D image. Figure 2.10 shows a 3D point being projected onto the projection window with a perspective projection.

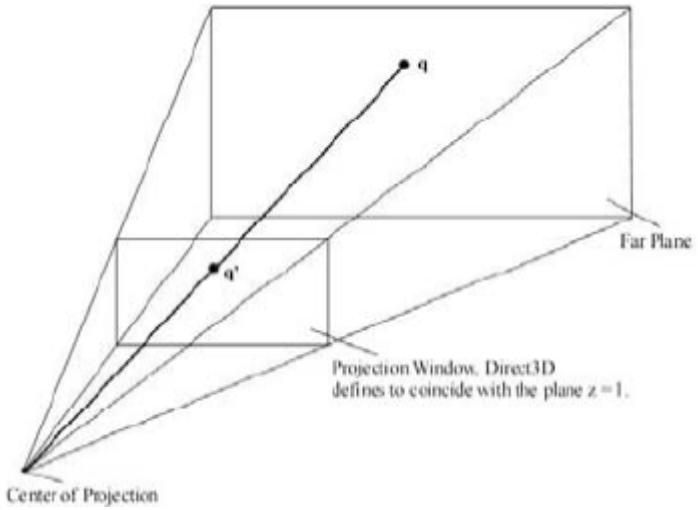


Figure 2.10: Projection

The projection transformation defines our viewing volume (frustum) and is responsible for projecting the geometry in the frustum onto the projection window.
We use the following D3DX function to create a projection matrix based on a frustum description:

```
D3DXMATRIX *D3DXMatrixPerspectiveFovLH(
      D3DXMATRIX* pOut, // returns projection matrix
      FLOAT fovY, // vertical field of view angle in radians
      FLOAT Aspect, // aspect ratio = width / height
      FLOAT zn, // distance to near plane
      FLOAT zf // distance to far plane
);
```

The aspect ratio is for correcting the stretching distortion caused when the projection window (a square) is transformed to the screen (a rectangle).
The projection matrix is set with the `IDirect3DDevice9::SetTransform` method, passing `D3DTS_PROJECTION` as the transform type. The following example creates a projection matrix based on a frustum with a 90-degree field of view, a near plane with a distance of 1, and a far plane with a distance of 1000.

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

2.3.7 Viewport Transform

The viewport transform is responsible for transforming coordinates on the projection window to a rectangle on the screen, which we call the *viewport*. For games, the viewport is usually the entire screen rectangle. In windowed mode it can be a subset of the screen or client area if we are running. The viewport rectangle is described relative to the window it resided in and is specified in window coordinates.

A viewport is represented by the `D3DVIEWPORT9` structure:

```
typedef struct _D3DVIEWPORT9 {
     DWORD X; // pixel coordinate of the upper-left corner of
              // the viewport on the render-target surface
     DWORD Y; //pixel coordinate
     DWORD Width;
     DWORD Height;
     float MinZ; // minimum depth buffer value
     float MaxZ; // maximum depth buffer value
} D3DVIEWPORT9;
```

The first four data members define the viewport rectangle relative to the window in which it resides. Direct3D uses a depth buffer range of zero to one, so MinZ and MaxZ should be set to those values respectively unless a special effect is desired.
The viewport can be set this way:

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

2.3.8 Rasterization

After the vertices are transformed to screen coordinates, we have a list of 2D triangles. The rasterization stage is responsible for computing the individual pixel color values needed to draw each triangle (see Figure 2.11).

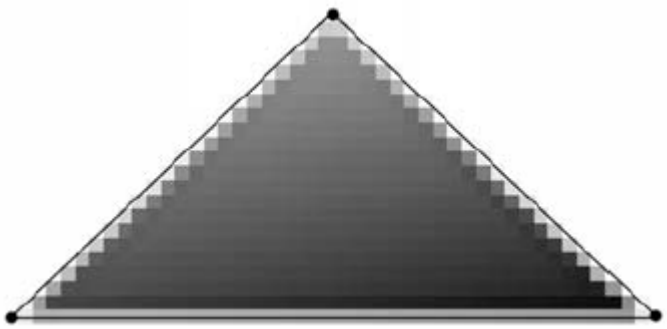


Figure 2.11: A triangle rasterized

The rasterization process is very intensive computationally and should always be done by dedicated graphics hardware. The end result of the rasterization stage is the 2D image that is displayed by the monitor.

2.4 Summary

- 3D objects are represented as triangle meshes, a list of triangles that approximates the shape and contours of the object.
- The virtual camera is modeled as a frustum. The volume of space inside the frustum is what the camera “sees”.
- 3D objects are defined in local space and are then all brought into one world space system. To facilitate projection, culling, and other operations, the objects are then transformed to view space, where the camera is positioned at the origin and looking down the positive z-axis. Once in view space, the objects are projected to the projection window. The viewport transformation transforms the geometry on the projection window to the viewport. Finally, the rasterization stage computes the individual pixel colors of the final 2D image.

3. Drawing

In this chapter, we will learn how to draw some geometric objects in Direct3D. The subjects from chapter 2 will be put into practice.

Keywords: vertex buffer, index buffer, `Lock`, `Unlock`, render states, drawing preparations, `DrawPrimitive`, `DrawIndexedPrimitive`, `BeginScene`, `EndScene`, D3DX geometry objects, `DrawSubset`.

3.1 Vertex/Index Buffers

A vertex buffer is a chunk of contiguous memory that contains vertex data. Similarly, an index buffer is a chunk of contiguous memory that contains index data. These buffers can be put in video memory for faster rendering. A vertex buffer is represented by the `IDirect3DVertexBuffer9` interface, the index buffer by `IDirect3DIndexBuffer9`.

3.1.1 Creating a Vertex and Index Buffer

We can create a vertex and index buffer with the following two methods:

```
HRESULT IDirect3DDevice9::CreateVertexBuffer(
      UINT Length,
      DWORD Usage,
      DWORD FVF,
      D3DPOOL Pool
      IDirect3DVertexBuffer9** ppVertexBuffer,
      HANDLE* pSharedHandle
);
HRESULT IDirect3DDevice9::CreateIndexBuffer(
      UINT Length,
      DWORD Usage,
      D3DFORMAT Format,
      D3DPOOL Pool,
      IDirect3DIndexBuffer9** ppIndexBuffer,
      HANDLE* pSharedHandle
);
```

The majority of the parameters are identical for both methods, so the parameters of both methods are described together.

- `Length`: The number of bytes to allocate for the buffer. If we wanted a vertex buffer to have enough memory to store eight vertices, we would set this parameter to 8 * sizeof(Vertex), where Vertex is our vertex structure.
- `Usage`: Specifies some additional properties about how the buffer is used. This value can be zero, indicating no additional properties, or a combination of one or more of the following flags:
    - `D3DUSAGE_DYNAMIC`: Setting this flag makes the buffer dynamic. See the notes on static and dynamic buffers on the following page.
    - `D3DUSAGE_POINTS`: This flag specifies that the buffer will hold point primitives. Point primitives are covered in “Particle Systems” in Chapter 14. This flag is used only for vertex buffers.
    - `D3DUSAGE_SOFTWAREPROCESSING`: Vertex processing is done in software.
    - `D3DUSAGE_WRITEONLY`: The application will only write to the buffer. This allows the driver to place the buffer in the best memory location for write operations. Reading from a buffer created with this flag will result in an error.
- `FVF`: The flexible vertex format of the vertices that is stored in the vertex buffer
- `Pool`: The memory pool in which the buffer is placed

- `ppVertexBuffer`: Pointer to receive the created vertex buffer
- `pSharedHandle`: Not used; set to zero
- `Format`: Specifies the size of the indices; use `D3DFMT_INDEX16` for 16-bit indices or use `D3DFMT_INDEX32` for 32-bit indices. Note that not all devices support 32-bit indices; check the device capabilities.
- `ppIndexBuffer`: Pointer to receive the created index buffer

A buffer created without the D3DUSAGE_DYNAMIC flag is called a static buffer. Static buffers are generally placed in video memory where its contents can be processed most efficiently. However, reading from the memory of a static buffer is slow because accessing video memory is slow. We use static buffers to hold data that will not need to be changed very frequently. Examples are terrains and city buildings. Static buffers should be filled with geometry at application initialization time and never at run time.
A buffer created with the D3DUSAGE_DYNAMIC flag is called a dynamic buffer. Dynamic buffers are generally placed in AGP memory where its memory can be updated quickly. Dynamic buffers are not processed as quickly as static buffers because the data must be transferred to video memory before rendering, but they can be updated reasonably fast. Therefore, if you need to update the contents of a buffer frequently, it should be made dynamic. Particle systems are good candidates for dynamic buffers because they are animated, and thus their geometry is usually updated every frame.

The following example creates a static vertex buffer with enough memory for eight vertices of type Vertex.

```
IDirect3DVertexBuffer9* vb;

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

3.1.2 Accessing a buffer's memory

To access the memory of a vertex/index buffer, we need to obtain a pointer to it with the `Lock` method. It is important to use `Unlock` when done accessing the memory. The parameters for both `Lock` methods are the same.

```
HRESULT IDirect3DVertexBuffer9::Lock(
     UINT OffsetToLock, //Offset, in bytes, from the start of the
                        //buffer to the location to begin the lock.
     UINT SizeToLock, //Number of bytes to lock
     BYTE** ppbData, //A pointer to the start of the locked memory
     DWORD Flags //Flags describing how the lock is done. See below
);

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

Setting both OffsetToLock and SizeToLock to zero means locking the entire buffer.
The Flags can be zero or one of the following:

- `D3DLOCK_DISCARD`: This flag is used only for dynamic buffers. It instructs the hardware to discard the buffer and return a pointer to a newly allocated buffer. This is

useful because it allows the hardware to continue rendering from the discarded buffer while we access the newly allocated buffer. This prevents the hardware from stalling.

- `D3DLOCK_NOOVERWRITE`: This flag is used only for dynamic buffers. It states that you are only going to append data to a buffer. That is, you will not overwrite any memory that is currently being rendered. This is beneficial because it allows the hardware to continue rendering at the same time you add new data to the buffer.
- `D3DLOCK_READONLY`: This flag states that you are locking the buffer only to read data and that you won't be writing to it. This allows for some internal optimizations.

The following example shows the `Lock` method being used:

```
Vertex* vertices;
_vb->Lock(0, 0, (void**)&vertices, 0); // lock the entire buffer

vertices[0] = Vertex(-1.0f, 0.0f, 2.0f); // write vertices to
vertices[1] = Vertex( 0.0f, 1.0f, 2.0f); // the buffer
vertices[2] = Vertex( 1.0f, 0.0f, 2.0f);

_vb->Unlock(); // unlock when you're done accessing the buffer
```

3.2 Render States

Direct3D encapsulates a variety of rendering states that affect how geometry is rendered. To set the render state to something other than the default, use:

```
HRESULT IDirect3DDevice9::SetRenderState(
      D3DRENDERSTATETYPE State, // the state to change
      DWORD Value // value of the new state
);
```

Example of setting the render state to wireframe mode:

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

3.3 Drawing preparations

Once we have created a vertex buffer and, optionally, an index buffer, we are almost ready to render its contents, but there are three steps that must be taken first.

Step 1. Set the stream source. Setting the stream source hooks up a vertex buffer to a stream that essentially feeds geometry into the rendering pipeline.
The following method is used to set a stream source:

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

- `StreamNumber`: Identifies the stream source to which we are hooking the vertex buffer. In this tutorial we always use stream zero.
- `pStreamData`: A pointer to the vertex buffer that we want to hook up to the stream
- `OffsetInBytes`: An offset from the start of the stream, measured in bytes, that specifies the start of the vertex data to be fed into the rendering pipeline. To set this

parameter to something besides zero, check if your device supports it by checking the `D3DDEVCAPS2_STREAMOFFSET` flag in the `D3DCAPS9` structure.

- `Stride`: Size in bytes of each element in the vertex buffer that we are attaching to the stream For example, suppose `vb` is a vertex buffer that has been filled with vertices of type `Vertex`:

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

Step 2. Set the vertex format. This is where we specify the vertex format of the vertices that we use in subsequent drawing calls.

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

Step 3. Set the index buffer. If we are using an index buffer, we must set the index buffer that is used in subsequent drawing operations. Only one index buffer can be used at a time; therefore if you need to draw an object with a different index buffer, you must switch to the other. The following code sets an index buffer:

```
_device->SetIndices( _ib ); // pass copy of index buffer pointer
```

3.4 Drawing with Vertex/Index Buffers

We can draw our geometry using `DrawPrimitive` or `DrawIndexedPrimitive`. These methods obtain the vertex info from the vertex streams and the index info from the currently set index buffer. The difference between `DrawPrimitive` and `DrawIndexedPrimitive` is that the first one does not use index info.

3.4.1 IDirect3DDevice9::DrawPrimitive

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

- `PrimitiveType`: The type of primitive that we are drawing. instance, we can draw points and lines in addition to triangles. Since we are using a triangle, use `D3DPT_TRIANGLELIST` parameter.
- `StartVertex`: Index to an element in the vertex streams marks the starting point from which to begin reading vertices. parameter gives us the flexibility to only draw certain portions of a vertex buffer.
- `PrimitiveCount`: The number of primitives to draw

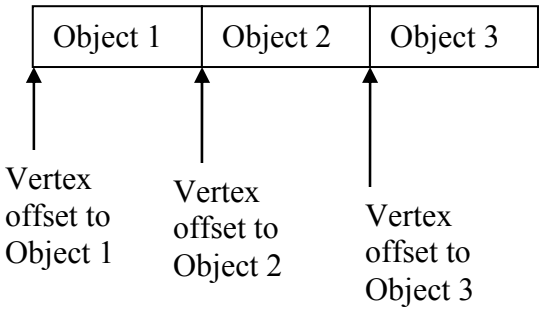
3.4.2 IDirect3DDevice9::DrawIndexedPrimitive

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

- `Type`: The type of primitive that we are drawing. For instance, we can draw points and lines in addition to triangles. Since we are using a triangle, use `D3DPT_TRIANGLELIST` for this parameter.

- `BaseVertexIndex`: A base number to be added to the indices used in this call. See below.
- `MinIndex`: The minimum index value that will be referenced
- `NumVertices`: The number of vertices that will be referenced in this call
- `StartIndex`: Index to an element in the index buffer that marks the starting point from which to begin reading indices
- `PrimitiveCount`: The number of primitives to draw

More on the `BaseVertexIndex`: Suppose that we want to combine the vertices three object into one global vertex buffer. For each object we would have to recompute the indices to index correctly into the global vertex buffer. The new indices are computed by adding an offset value that specifies the start of the object's vertices in the global vertex buffer to each index. Note that the offset is measured in vertices, not bytes.
Rather than having to compute the indices relative to where the object is in the global vertex buffer by ourselves, Direct3D allows us to pass in a vertex-offset value through the BaseVertexIndex parameter. Direct3D will then recompute the indices internally.



3.4.3 Begin/End Scene

All drawing methods must be called inside an `IDirect3DDevice9::BeginScene` and `IDirect3DDevice9::EndScene` pair. For example:

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

This is like the OpenGL's `glBegin()` and `glEnd()` pair.

3.5 D3DX Geometry Objects

The D3DX library provides some methods to generate the mesh data of simple 3D objects. These are: `D3DXCreateBox, D3DXCreateSphere, D3DXCreateCylinder, D3DXCreateTeapot, D3DXCreatePolygon` and `D3DXCreateTorus.`
All six are used similarly and use the D3DX mesh data structure `ID3DXMesh` as well as the `ID3DXBuffer` interface. For now, we ignore their details.
The `D3DXCreateTeapot` function:

```
HRESULT D3DXCreateTeapot(
      LPDIRECT3DDEVICE9 pDevice, // device associated with the mesh
      LPD3DXMESH* ppMesh, // pointer to receive mesh
      LPD3DXBUFFER* ppAdjacency // set to zero for now
);
```

Example of its usage:

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

The generated mesh can be rendered with the `ID3DXMesh::DrawSubset` method. Make sure to call `Release` when done with the mesh.

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

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

3.6 Examples

There are some example applications for subjects from this chapter. They can be found at part II chapter 3. These examples include Triangle, Cube, Teapot and D3DXCreate. The latter two use the D3DX library to generate the teapot and other objects.
I will briefly explain how each example works. They can be studied on your own for more information.
All files use d3dUtility.cpp and d3dUtility.h for handling Direct3D initialization. We will only concentrate on the methods `Setup`, `Display` and `Release` for each of the examples.

3.6.1 Triangle

The `Setup` method for Triangle contains code for instantiating a Vertex buffer called `IDirect3DVertexBuffer9* Triangle`.

The code for generating a simple triangle:

```
Vertex* vertices;
Triangle->Lock(0, 0, (void**)&vertices, 0);

vertices[0] = Vertex(-1.0f, 0.0f, 2.0f);
vertices[1] = Vertex( 0.0f, 1.0f, 2.0f);
vertices[2] = Vertex( 1.0f, 0.0f, 2.0f);

Triangle->Unlock();
```

This buffer is released in the `Cleanup` method.

```
d3d::Release<IDirect3DVertexBuffer9*>(Triangle);
```

The triangle is set and drawn in the Display method using:

```
Device->SetStreamSource(0, Triangle, 0, sizeof(Vertex)); // Step 1
Device->SetFVF(Vertex::FVF); // Step 2
Device->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);
```

This is done between the `BeginScene()` and `EndScene()` methods.

3.6.2 Cube

The Cube example is very similar to Triangle. Except that Cube also uses an Index buffer. Both buffers are defined as:

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

They are of course instantiated in the `Setup` method. The `Display` method displays the cube using these calls:

```
Device->BeginScene();

Device->SetStreamSource(0, VB, 0, sizeof(Vertex)); // Step 1
Device->SetIndices(IB); // Step 3
Device->SetFVF(Vertex::FVF); // Step 2

Device->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0, 8, 0, 12);

Device->EndScene();
```

It first sets the Vertex buffer `VB` and then the Index buffer `IB`. Finally the drawing is done using `DrawIndexedPrimitive`. Step 2 and 3 were swapped, but that doesn't matter.

3.6.3 Teapot and D3DXCreate

Both examples use the D3DX library to generate object(s). The methods to create D3DX geometry objects were listed at chapter 3.5 (`D3DXCreateTeapot` for example). An object is stored in an `ID3DXMesh` pointer and drawn with `DrawSubset`.
Example code for the Teapot example:

```
// Global variable
ID3DXMesh* Teapot = 0;

// Display() method
Device->BeginScene();

// Draw teapot using DrawSubset method with 0 as the argument.
Teapot->DrawSubset(0);

Device->EndScene();
```

The D3DXCreate sample has the same structure, but draws more geometry objects.

3.7 Summary

- Vertex data is stored in the `IDirect3DVertexBuffer9` interface. Similarly, index data is stored in the `IDirect3DIndexBuffer9` interface. The reason for using vertex/index buffers is that the data can be stored in video memory.
- Geometry that is static should be stored in a static vertex/index buffer. Geometry that is dynamic should be stored in a dynamic vertex/index buffer.
- Render states are states that the device maintains that affect how geometry is rendered. Render states remain in effect until changed, and the current values are applied to the geometry of any subsequent drawing operations. All render states have initial default values.
- To draw the contents of a vertex buffer and an index buffer you must:
    - Call `IDirect3DDevice9::SetStreamSource` and hook the vertex buffer that you wish to draw from to a stream.
    - Call `IDirect3DDevice9::SetFVF` to set the vertex format of the vertices to render.

    - If you are using an index buffer, call `IDirect3DDevice9::SetIndices` to set the index buffer.
    - Call either `IDirect3DDevice9::DrawPrimitive` or `IDirect3DDevice9::DrawIndexedPrimitive` in between an `IDirect3DDevice9::BeginScene` and `IDirect3DDevice9::EndScene` pair.
- Using the `D3DXCreate*` functions, we can create complex 3D objects, such as spheres, cylinders, and teapots.

4. Color

4.1 Color Representation

Colors in Direct3D are described with an RGB triplet (red, green and blue). We can use `D3DCOLOR` or the struct `D3DCOLORVALUE` to hold RGB data.
Keywords: RGB, `D3DCOLOR, D3DCOLORVALUE`, vertex colors, shading.

4.1.1 D3DCOLOR

`D3DCOLOR` is actually a `DWORD` (`typedef`ed `unsigned long`) and is 32 bits. The bits are divided into four 8-bit sections, making each section range from 0-255. Near 0 means low intensity and near 255 means strong intensity. The sections are divided for alpha, red, green and blue. The alpha component is used for alpha blending and will be used in chapter 7, ignore it for now.
Direct3D provides two macros to specify and insert each component into a `D3DCOLOR` type. Namely `D3DCOLOR_ARGB` and `D3DCOLOR_XRGB`. There is a parameter for each color and one for the alpha component. `D3DCOLOR_XRGB` is the same, but without the alpha component. That component will be set to 0xff (255).

```
//D3DCOLOR ARGB(a,r,g,b)
D3DCOLOR red = D3DCOLOR_ARGB(255, 255, 0, 0);
D3DCOLOR someColor = D3DCOLOR_ARGB(255, 144, 87, 201);

//#define D3DCOLOR_XRGB(r,g,b) D3DCOLOR_ARGB(0xff,r,g,b)
D3DCOLOR brightGreen = D3DCOLOR_XRGB(0, 255, 0);
D3DCOLOR someColor2 = D3DCOLOR_XRGB(144, 87, 201);
```

4.1.2 D3DCOLORVALUE

This is a structure with the same four components. But each is specified with a float, ranging from 0 (no intensity) to 1 (full intensity).

```
typedef struct _D3DCOLORVALUE {
    float r; // the red component, range 0.0-1.0
    float g; // the green component, range 0.0-1.0
    float b; // the blue component, range 0.0-1.0
    float a; // the alpha component, range 0.0-1.0
} D3DCOLORVALUE;
```

4.1.3 D3DXCOLOR

Another way to store RGB data is using the `D3DXCOLOR` structure, which contains the same data members as `D3DCOLORVALUE` but provides useful constructors and overloaded operators. This makes color operations easy and makes casting between `D3DCOLOR` and `D3DCOLORVALUE` possible.

```
typedef struct D3DXCOLOR {
#ifdef   cplusplus
public:
     D3DXCOLOR() {}
     D3DXCOLOR( DWORD argb );
     D3DXCOLOR( CONST FLOAT * );
     D3DXCOLOR( CONST D3DXFLOAT16 * );
     D3DXCOLOR( CONST D3DCOLORVALUE& );
     D3DXCOLOR( FLOAT r, FLOAT g, FLOAT b, FLOAT a );
     // casting
     operator DWORD () const;
     operator FLOAT* ();
     operator CONST FLOAT* () const;
     operator D3DCOLORVALUE* ();
     operator CONST D3DCOLORVALUE* () const;
     operator D3DCOLORVALUE& ();
     operator CONST D3DCOLORVALUE& () const;
     // assignment operators
     D3DXCOLOR& operator += ( CONST D3DXCOLOR& );
     D3DXCOLOR& operator -= ( CONST D3DXCOLOR& );
     D3DXCOLOR& operator *= ( FLOAT );
     D3DXCOLOR& operator /= ( FLOAT );
     // unary operators
     D3DXCOLOR operator + () const;
     D3DXCOLOR operator - () const;
     // binary operators
     D3DXCOLOR operator + ( CONST D3DXCOLOR& ) const;
     D3DXCOLOR operator - ( CONST D3DXCOLOR& ) const;
     D3DXCOLOR operator * ( FLOAT ) const;
     D3DXCOLOR operator / ( FLOAT ) const;
     friend D3DXCOLOR operator * (FLOAT, CONST D3DXCOLOR& );
     BOOL operator == ( CONST D3DXCOLOR& ) const;
     BOOL operator != ( CONST D3DXCOLOR& ) const;
#endif //  cplusplus
     FLOAT r, g, b, a;
} D3DXCOLOR, *LPD3DXCOLOR;
```

Please note that the examples now have some colors added to the d3dUtility.h file. Like:

```
const D3DXCOLOR WHITE( D3DCOLOR_XRGB(255, 255, 255) );
const D3DXCOLOR BLACK( D3DCOLOR_XRGB( 0, 0, 0) );
```

4.2 Vertex Colors

The color of a primitive is determined by the color of the vertices that make it up. Therefore, we must add a color member to our vertex data structure. Direct3D expects a 32-bit value to describe the color of a vertex, so we must use `D3DCOLOR` and not `D3DCOLORVALUE`.

```
struct ColorVertex
{
     float  x,  y,  z;
     D3DCOLOR _color;
     static const DWORD FVF;
}
const DWORD ColorVertex::FVF = D3DFVF_XYZ | D3DFVF_DIFFUSE;
```

4.3 Shading

Shading occurs during rasterization and specifies how the vertex colors are used to compute the pixel colors that make up the primitive. We can use flat shading or Gouraud shading. Take a look at this code:

```
ColorVertex t[3];
t[0]. color = D3DCOLOR XRGB(255, 0, 0);
t[1]. color = D3DCOLOR XRGB(0, 255, 0);
t[2]._color = D3DCOLOR_XRGB(0, 0, 255);
```

With flat shading, the color specified in the first vertex is taken. The first color here is red, so the triangle formed with these vertices will be red. Flat shading tends to make objects appear blocky.
With Gouraud shading, the colors at each vertex are interpolated linearly across the face of the primitive. This leads to a "smoother" object. A sample application called Colored Triangle will be shown in section 4.4 to see the difference between the two shading modes. The shading can be set with:

```
// set flat shading
Device->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_FLAT);
// set Gouraud shading
Device->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_GOURAUD);
```

4.4 Sample: Colored Triangle

This example shows a triangle colored using flat shading and a triangle colored using Gouraud shading.
First, the triangle is created using a vertex buffer in the `Setup` method:

```
// create vertex buffer
Device->CreateVertexBuffer(
      3 * sizeof(ColorVertex),
      D3DUSAGE_WRITEONLY,
      ColorVertex::FVF,
      D3DPOOL_MANAGED,
      &Triangle, // IDirect3DVertexBuffer9* Triangle
      0);

// fill the buffers with the triangle data
ColorVertex* v; // ColorVertex struct was defined in 4.2
Triangle->Lock(0, 0, (void**)&v, 0);

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

Triangle->Unlock();
```

The actual drawing in the `Display` method:

```
Device->BeginScene();

Device->SetFVF(ColorVertex::FVF); // Step 2, see chapter 3.3
Device->SetStreamSource(0,Triangle,0,sizeof(ColorVertex)); // Step 1

// draw the triangle to the left with flat shading
D3DXMatrixTranslation(&WorldMatrix, -1.25f, 0.0f, 0.0f);
Device->SetTransform(D3DTS_WORLD, &WorldMatrix);
Device->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_FLAT);
Device->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);

// draw the triangle to the right with Gouraud shading
D3DXMatrixTranslation(&WorldMatrix, 1.25f, 0.0f, 0.0f);
```

```
Device->SetTransform(D3DTS_WORLD, &WorldMatrix);
Device->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_GOURAUD);
Device->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);

Device->EndScene();
```

Only one vertex buffer was needed for the triangle, because we can draw it multiple times at different positions using the global variable `D3DXMATRIX WorldMatrix`.

4.5 Summary

- Colors are described by specifying an intensity of red, green, and blue (RGB). In Direct3D, we can use the `D3DCOLOR`, the `D3DCOLORVALUE`, or the `D3DXCOLOR` type to describe a color in code.
- We sometimes treat a color as a 4D vector (r, g, b, a). Color vectors are added, subtracted, and scaled just like regular vectors. On the other hand, dot and cross products do not make sense for color vectors, but component-wise multiplication does make sense for colors: $(c_1, c_2, c_3, c_4) * (k_1, k_2, k_3, k_4) = (c_1k_1, c_2k_2, c_3k_3, c_4k_4)$.
- With flat shading, the pixels of a primitive are uniformly colored by the color specified in the first vertex of the primitive. With Gouraud shading, the colors at each vertex are interpolated linearly across the face of the primitive.

5. Lighting

In this chapter we will discuss adding lighting to the scenes. When using lighting, we no longer specify vertex colors ourselves. Direct3D computes a vertex color based on defined light sources, materials, and the orientation of the surface with regard to the light sources.
Keywords: ambient, diffuse, specular, emissive, material, `D3DMATERIAL9`, face normal, vertex normal, normal averaging, point, directional, spot.

5.1 Light Components

There are four kinds of light in the Direct3D lighting model.

- Ambient Light: This light is used to bright up the overall scene. It is constant in all directions and it colors all pixels of an object the same. Parts of objects are often lit, to a degree, even though they are not in direct sight of a light source. Without ambient light, objects in shadow would be completely black.

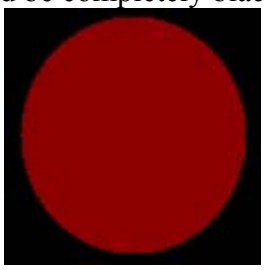


Figure 5.1: Ambient lighted model

- Diffuse Light: This type of light travels in a particular direction. When it strikes a surface, it reflects equally in all directions. Because diffuse light reflects equally in all directions, the reflected light will reach the eye no matter the viewpoint.

Figure 5.2a: Model with only diffuse light from the left

Figure 5.2b: Same model with ambient and diffuse light combined

- Specular Light: This type of light travels in a particular direction. When it strikes a surface, it reflects harshly in one direction, causing a bright shine that can only be seen from some angles. Specular light is used to model light that produces highlights on objects, such as the bright shine created when light strikes a polished surface.

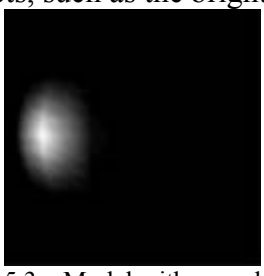


Figure 5.3a: Model with specular light

Figure 5.3b: Ambient, diffuse and specular light combined.

- Emissive Lighting: Is emitted by an object; for example, a glow.

Specular lighting requires the most computations. It is turned off by default. Set the `D3DRS_SPECULARENABLE` render state to enable specular lighting.

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

Each type of light is represented by a `D3DCOLORVALUE` structure or `D3DXCOLOR`, which describes the color of the light. Here are some examples of several light colors:

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

5.2 Materials

The color of an object we see in the real world is determined by the color of light that the object reflects. For instance, a red ball is red because it absorbs all colors of light except red light. Direct3D models this by defining a material for an object, which allows us to define the percentage at which light is reflected from the surface. The `D3DMATERIAL9` structure is used for this.

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

With this struct you can specify the amount of each light the surface reflects. The `Power` parameter specifies the sharpness of specular highlights; the higher this value, the sharper the highlights.
As an example, for a red ball we would define the ball's material to reflect only red light and absorb all other colors of light:

```
D3DMATERIAL9 red;
::ZeroMemory(&red, sizeof(red));
red.Diffuse = D3DXCOLOR(1.0f, 0.0f, 0.0f, 1.0f); // red
red.Ambient = D3DXCOLOR(1.0f, 0.0f, 0.0f, 1.0f); // red
red.Specular = D3DXCOLOR(1.0f, 0.0f, 0.0f, 1.0f); // red
red.Emissive = D3DXCOLOR(0.0f, 0.0f, 0.0f, 1.0f); // no emission
red.Power = 5.0f;
```

With a light source that only emits blue light, this ball would appear black because it absorbs all the light.
The following function is added to the d3dUtility.h/cpp files to help us fill out a material structure:

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

To set the current material, we use the following method:

```
IDirect3DDevice9::SetMaterial(CONST D3DMATERIAL9*pMaterial)
```

To render several objects with different materials, call this method with a different material before drawing each object:

```
D3DMATERIAL9 blueMaterial, redMaterial;
...// set up material structures

Device->SetMaterial(&blueMaterial);
drawSphere(); // blue sphere

Device->SetMaterial(&redMaterial);
drawSphere(); // red sphere
```

5.3 Vertex Normals

A *face normal* is a vector that describes the direction a polygon is facing. *Vertex normals* are based on the same idea, but rather than specifying the normal per polygon, we specify them for each vertex that forms the polygon.

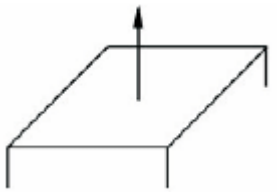


Figure 5.4a: face normal

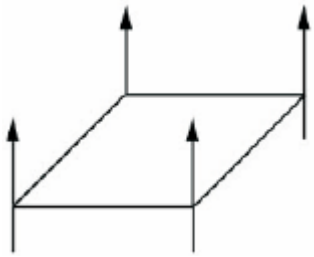


Figure 5.4b: vertex normals

Direct3D needs to know the vertex normals so that it can determine the angle at which light strikes a surface, and since lighting calculations are done per vertex, Direct3D needs to know the surface orientation (normal) per vertex.
Our custom vertex structure from 4.2 is now updated to include normals:

```
struct Vertex {
      float  x,  y,  z;
      float  nx,  ny,  nz;
      static const DWORD FVF;
}
const DWORD Vertex::FVF = D3DFVF_XYZ | D3DFVF_NORMAL;
```

The color member is no longer needed because the color of each vertex is now calculated using lighting.

To compute the normals for complex meshes, the following way is the simplest. Suppose a triangle that is formed by three vertices, the normals will be the face normal these three vertices form. The following method computes this face normal. It assumes that the vertices are specified in a clockwise winding order.

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

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

The normal of the vertices `p0`, `p1` and `p2` will be the result of this method, namely `out`.

A problem with face normals is that it does not produce smooth results. *Normal averaging* would be a better method for finding a vertex normal. To find the vertex normal $\mathbf{v_n}$ of a vertex $\mathbf{v}$, we find the face normals for all the triangles in the mesh that share vertex $\mathbf{v}$. Then $\mathbf{v_n}$ is given by averaging all of these face normals.
Suppose three triangles share the vertex $\mathbf{v}$. Their face normals are $\mathbf{n_0}$, $\mathbf{n_1}$, and $\mathbf{n_2}$,.
Then the vertex normal $\mathbf{v_n}$ is given by averaging the face normals:
$\mathbf{v_n} = \frac{1}{3} (\mathbf{n_0} + \mathbf{n_1} + \mathbf{n_2})$

During the transformation stages, it is possible for vertex normals to become non-normal. Therefore, it is best to be safe and have Direct3D renormalize all of your normals after the transformation stages by enabling the `D3DRS_NORMALIZENORMALS` render state:

```
Device->SetRenderState(D3DRS_NORMALIZENORMALS, true);
```

5.4 Light Sources

Direct3D support the following types of light sources:
Point lights: has a position in world space and emits light in all directions.
Directional lights: has no position but shoots parallel rays of light in the specified direction.

Spot lights: is similar to a flashlight; it has a position and shines light through a conical shape in a particular direction. The angle φ (phi) describes an inner cone, and θ (theta) describes the outer cone.
A light source is represented by the `D3DLIGHT9` structure:

```
typedef struct  D3DLIGHT9 {
      D3DLIGHTTYPE Type;
      D3DCOLORVALUE Diffuse;
      D3DCOLORVALUE Specular;
      D3DCOLORVALUE Ambient;
      D3DVECTOR Position;
      D3DVECTOR Direction;
      float Range;
      float Falloff;
      float Attenuation0;
      float Attenuation1;
      float Attenuation2;
      float Theta;
      float Phi;
} D3DLIGHT9;
```

The `Type` field defines the type of light and can be `D3DLIGHT_POINT`, `D3DLIGHT_SPOT` or `D3DLIGHT_DIRECTIONAL`.
For information on the other fields of this struct, search for `D3DLIGHT9` in the SDK documentation.

The following function is added to the d3dUtility.h/cpp files to help us initialize a `D3DLIGHT9` structure:

```
D3DLIGHT9 InitDirectionalLight(D3DXVECTOR3* direction,
                               D3DXCOLOR* color);
D3DLIGHT9 InitPointLight(D3DXVECTOR3* position,
                         D3DXCOLOR* color);
D3DLIGHT9 InitSpotLight(D3DXVECTOR3* position,
                         D3DXVECTOR3* direction,
                         D3DXCOLOR* color);
```

Then to create a directional light that runs parallel with the x-axis in the positive direction and emits white light, we would write:

```
D3DXVECTOR3 dir(1.0f, 0.0f, 0.0f);
D3DXCOLOR c = d3d::WHITE;
D3DLIGHT9 dirLight = d3d::InitDirectionalLight(&dir, &c);
```

After we have initialized a `D3DLIGHT9` instance, we need to register with an internal list of lights that Direct3D maintains:

```
Device->SetLight(
```

```
      0, // element in the light list to set, range is 0-maxlights
      &light); // address of the D3DLIGHT9 structure to set
```

Once a light is registered, we can turn it on and off:

```
Device->LightEnable(
      0, // the element in the light list to enable/disable
      true); // true = enable, false = disable
```

5.5 Sample: litPyramid

The example litPyramid can be found at part 2 chapter 5. It shows a rotating pyramid with a light.

The steps for adding light to a scene are:
Step 1. Enable lighting.
Step 2. Create a material for each object and set the material before rendering the corresponding object.
Step 3. Create one or more light sources, set the light sources, and enable them.
Step 4. Enable any additional lighting states, such as specular highlights.

Only the most interesting code of the `Setup` method is shown here:

```
/* STEP 1: Turn on lighting. It is enabled by default, but this
won't hurt */
Device->SetRenderState(D3DRS LIGHTING, true);
.
.
// In this part the Pyramid vertex buffer if created and filled
.
.
/* STEP 2: Create and set the material */
D3DMATERIAL9 mtrl;
mtrl.Ambient  = d3d::WHITE;
mtrl.Diffuse  = d3d::WHITE;
mtrl.Specular = d3d::WHITE;
mtrl.Emissive = d3d::BLACK;
mtrl.Power    = 5.0f;

/* STEP 3: Create one or more light sources, set the light sources,
and enable them */

// Setup a directional light
D3DLIGHT9 dir;
::ZeroMemory(&dir, sizeof(dir));
dir.Type      = D3DLIGHT DIRECTIONAL;
dir.Diffuse   = d3d::WHITE;
dir.Specular  = d3d::WHITE * 0.3f;
dir.Ambient   = d3d::WHITE * 0.6f;
dir.Direction = D3DXVECTOR3(1.0f, 0.0f, 0.0f);

// Set and Enable the light
Device->SetLight(0, &dir);
Device->LightEnable(0, true);

/* STEP 4: Enable any additional lighting states, such as specular
highlights */

/* Turn on specular lighting and instruct Direct3D to renormalize
normals */
```

```
Device->SetRenderState(D3DRS_NORMALIZENORMALS, true);
Device->SetRenderState(D3DRS_SPECULARENABLE, true);
.
.
```

There are additional samples provided for this subject. These samples use the `D3DXCreate*` functions to create 3D objects. The vertex normals will be computed for us. Also, they make use of the d3dUtility.h/cpp material and light functionality code. The first sample did not use these to show how it's done manually.

5.6 Summary

- Direct3D supports four light source models: directional lights, point lights, spot lights and emissive lights.
- The material of a surface defines how light interacts with the surface that it strikes (that is, how much light is reflected and absorbed, thus determining the color of the surface).
- Vertex normals are used to define the orientation of a vertex. They are used so that Direct3D can determine the angle at which a ray of light strikes the vertex. In some cases, the vertex normal is equal to the normal of the triangle that it forms, but this is not usually the case when approximating smooth surfaces (e.g., spheres, cylinders).

6. Texturing

Direct3D has the ability to map images onto the surfaces of our triangles so they can look like brick, metal, or whatever. This capability is called texturing. A texture is simply an image, say from a BMP file, which is mapped onto a triangle or series of triangles. Texturing increases the details and realism significantly. As an example, you could build a cube and turn it into a crate by mapping a crate texture to each side.

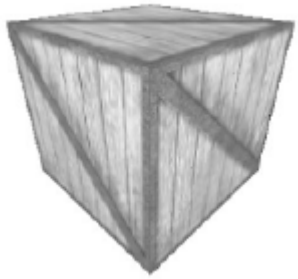


Figure 6.1: A cube with crate texture

Keywords: texture coordinates, texel, `IDirect3DTexture9`, filters, Nearest point sampling, Linear filtering, Anisotropic filtering, mipmap, mipmap filter, address modes

6.1 Texture Coordinates

To use textures, the custom vertex structure will need to be modified again:

```
struct Vertex
{
      float  x,  y,  z;
      float  nx,  ny,  nz;
      float  _u, _v; // texture coordinates

      static const DWORD FVF;
};
const DWORD Vertex::FVF = D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1;
```

We added `D3DFVF_TEX1` to the vertex format description, making clear that our vertex structure contains one pair of vertex coordinates.
Also note that texture coordinates are added. Texture coordinates are two-dimensional and range from 0.0 to 1.0, where (0.0, 0.0) represents the top-left side of the texture and (1.0, 1.0) represents the lower-right side of the texture. The horizontal axis of this coordinate system is called *u* and the vertical one is called *v*. A (*u, v*) pair is called a *texel*. See figure 6.2.

Figure 6.2: Texture coordinates

Figure 6.3: Result after mapping to (0,1), (0,0), (1,0)

If we map the texture onto a triangle using the coordinates (0,1), (0,0) and (1,0) the result will be like figure 6.3.

6.2 Using textures in code

The D3DX function `D3DXCreateTextureFromFile` can be used to load an image file into an `IDirect3DTexture9` object.

```
HRESULT D3DXCreateTextureFromFile(
      LPDIRECT3DDEVICE9 pDevice, // device to create the texture
      LPCSTR pSrcFile, // filename of image to load
      LPDIRECT3DTEXTURE9* ppTexture // ptr to receive the created
      texture
);
```

For example, to create a texture from an image called stonewall.bmp, we would write the following:

```
IDirect3Dtexture9*  _stonewall;
D3DXCreateTextureFromFile(_device, "stonewall.bmp", &_stonewall);
```

To set the current texture, use:

```
HRESULT IDirect3DDevice9::SetTexture(
      DWORD Stage, // the texture stage
      IDirect3DBaseTexture9* pTexture // ptr to the texture to set
);

//Example usage:
Device->SetTexture(0, _stonewall);
```

You can actually have more than one texture active at the same time, which is what the Stage argument is for. This is called *multitexturing* and is an advanced topic. Set it to 0 for now.

To disable a texture at a particular texturing stage, set pTexture to 0. For instance, if we don't want to render an object with a texture, we would write:

```
Device->SetTexture(0, 0);
renderObjectWithoutTexture();
```

If our scene has triangles that use different textures, we would have to do something similar to the following code:

```
Device->SetTexture(0,  tex0);
drawTrisUsingTex0();
Device->SetTexture(0,  tex1);
drawTrisUsingTex1();
```

6.3 Filters

When a texture triangle is smaller than the screen triangle, the texture triangle is magnified to fit. When a texture triangle is larger than the screen triangle, the texture triangle is minified. In both cases, distortion will occur. *Filtering* is a technique Direct3D uses to help smooth out these distortions.
IDirect3DDevice9::SetSamplerState is used to set to filter type. Each type has a different level of quality. The better the quality, the slower it is. The possible filter types are:

- **Nearest-point sampling**: Computes the texel address and copies the color of the texel with the closest integer address. This is the default filter type. It is also the fastest, but has the worst looking results. To set nearest-point sampling as the minification and magnification filter, use:

```
Device->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_POINT);
Device->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_POINT);
```

- **Linear filtering**: Is called bilinear filtering in the documentation. Does the same as nearest-point sampling, but in addition computes a weighted average of the texels that are immediately above, below, to the left of, and to the right of the nearest sample point.
  This filter type produces fairly good results and is very fast on today's hardware.
  Using linear filtering as minimum is recommended.

```
Device->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
Device->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
```

- **Anisotropic filtering**: Anisotropy is the distortion visible in the image of a 3-D object whose surface is oriented at an angle with respect to the viewing plane.
  Produces the best results, but the most computation intensive.

```
Device->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_ANISOTROPIC);
Device->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_ANISOTROPIC);
```

  The `D3DSAMP_MAXANISOTROPY` level must be set when using this type of filtering. The higher this level is, the better the quality. The `D3DCAPS9` structure indicates the range your device supports.
  Example for setting `D3DSAMP_MAXANISOTROPY` level to 4:

```
Device->SetSamplerState(0, D3DSAMP_MAXANISOTROPY, 4);
```

6.4 Mipmaps

Another solution for the distortion problem from 6.3 is using *mipmaps*. We take a texture and create a series of smaller lower resolution textures.

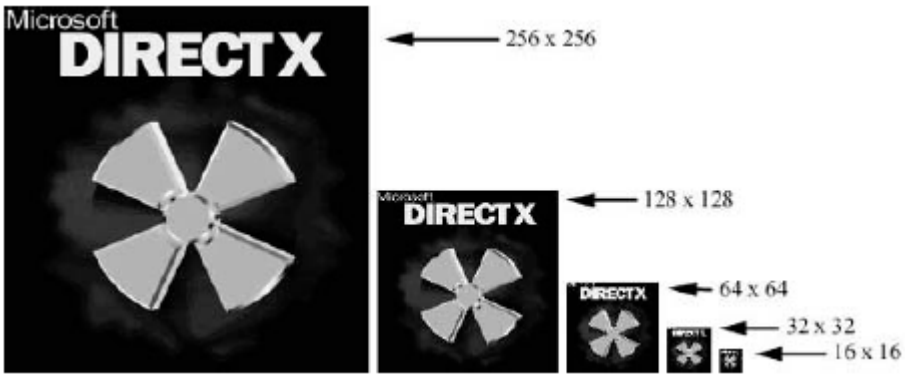


Figure 6.4: A chain of mipmaps

If the device supports mipmaps, `D3DXCreateTextureFromFile` will generate a mipmap chain for you. Direct3D will automatically select the mipmap that matches the screen triangle the best. You can also use the DirectX Texture Editor (DxTex.exe) to generate a mipmap chain for you. It can be found in "[YOUR_DXSDK_ROOT_DIR]\Utilities\Bin\x86" and at the Windows start menu: "Microsoft DirectX 9.0 SDK Update (June 2005) -> DirectX Utilities -> DirectX Texture Tool".
Search for "Mipmapping" in the SDK documentation to find out how generate a mipmap chain with the Texture Editor.

6.4.1 Mipmap filter

The mipmap filter is used to control how Direct3D uses the mipmaps. It can be set with:

```
Device->SetSamplerState(0, D3DSAMP_MIPFILTER, Filter);
```

The `Filter` option can be:

- `D3DTEXF_NONE`: Disables mipmapping
- `D3DTEXF_POINT`: By using this filter, Direct3D will choose the mipmap level that is closest in size to the screen triangle. Once that level is chosen, Direct3D will filter that level based on the specified minification and magnification filters.
- `D3DTEXF_LINEAR`: By using this filter, Direct3D will take the two closest mipmap levels, filter each level with the minification and magnification filters, and linearly combine these two levels to form the final color values.

6.5 Address Modes

Texture coordinates can exceed the range [0, 1]. The address modes must be set to control the behavior of Direct3D with texture coordinates outside this range. These address modes are wrap, border, clamp and mirror. See figure 6.5.

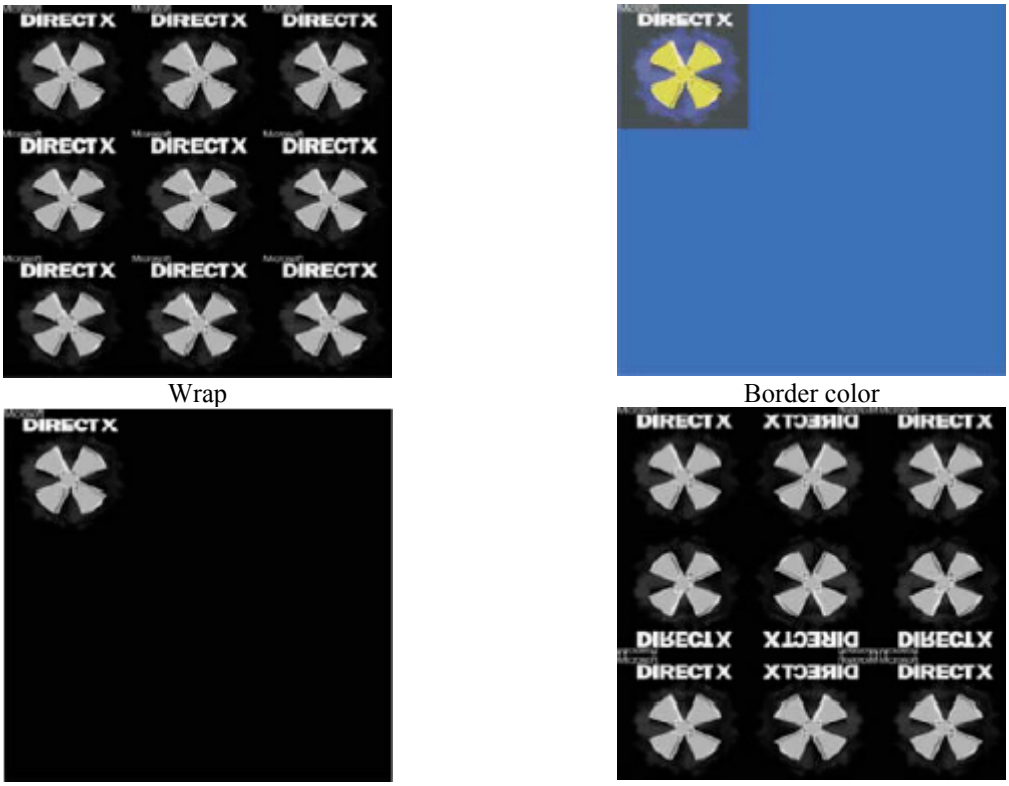


Figure 6.5: Address modes

The texture coordinates used in these figures are (0, 0), (0, 3), (3, 0), and (3, 3). This is why the quad is tiled into a 3 x 3 area.
Example of how the address modes are set:

```
// set wrap address mode
Device->SetSamplerState(0, D3DSAMP_ADDRESSU, D3DTADDRESS_WRAP);
Device->SetSamplerState(0, D3DSAMP_ADDRESSV, D3DTADDRESS_WRAP);

// set border color address mode
Device->SetSamplerState(0, D3DSAMP_ADDRESSU, D3DTADDRESS_BORDER);
Device->SetSamplerState(0, D3DSAMP_ADDRESSV, D3DTADDRESS_BORDER);
Device->SetSamplerState(0, D3DSAMP_BORDERCOLOR, 0x000000ff);

// set clamp address mode
Device->SetSamplerState(0, D3DSAMP_ADDRESSU, D3DTADDRESS_CLAMP);
Device->SetSamplerState(0, D3DSAMP_ADDRESSV, D3DTADDRESS_CLAMP);

// set mirror address mode
Device->SetSamplerState(0, D3DSAMP_ADDRESSU, D3DTADDRESS_MIRROR);
Device->SetSamplerState(0, D3DSAMP_ADDRESSV, D3DTADDRESS_MIRROR);
```

6.6 Sample: Textured Quad

The sample application Textured Quad shows how to take the steps needed for adding textures to a scene:

Step 1. Construct the vertices of the objects with the texture coordinates specified.

Step 2. Load a texture into an `IDirect3DTexture9` interface using the `D3DXCreateTextureFromFile` function.

Step 3. Set the minification, magnification, and mipmap filters.

Step 4. Before drawing an object, set the texture that is associated with the object with `IDirect3DDevice9::SetTexture`.

The sample takes the usual steps for drawing something, like creating a vertex buffer, drawing it at the `Display` method etc. I will only show the `Setup` method. The four steps shown above will be taken.

First the global variables:

```
IDirect3DVertexBuffer9* Quad = 0;
IDirect3DTexture9* Tex = 0;
```

Code snippet of this sample's `Setup` method:

```
Vertex* v;
Quad->Lock(0, 0, (void**)&v, 0);

// STEP 1
// quad built from two triangles, note texture coordinates:
v[0] = Vertex(-1.0f, -1.0f, 1.25f, 0.0f, 0.0f, -1.0f, 0.0f, 1.0f);
v[1] = Vertex(-1.0f, 1.0f, 1.25f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f);
v[2] = Vertex( 1.0f, 1.0f, 1.25f, 0.0f, 0.0f, -1.0f, 1.0f, 0.0f);
v[3] = Vertex(-1.0f, -1.0f, 1.25f, 0.0f, 0.0f, -1.0f, 0.0f, 1.0f);
v[4] = Vertex( 1.0f, 1.0f, 1.25f, 0.0f, 0.0f, -1.0f, 1.0f, 0.0f);
v[5] = Vertex( 1.0f, -1.0f, 1.25f, 0.0f, 0.0f, -1.0f, 1.0f, 1.0f);

Quad->Unlock();

// STEP 2: Load texture data.
D3DXCreateTextureFromFile(
     Device,
     "dx5_logo.bmp",
     &Tex);

// STEP 4: Enable the texture.
Device->SetTexture(0, Tex);

// STEP 3: Set texture filters.
Device->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
Device->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
Device->SetSamplerState(0, D3DSAMP_MIPFILTER, D3DTEXF_POINT);
```

6.7 Summary

- Texture coordinates are used to define a triangle on the texture that gets mapped to the 3D triangle.
- We can create textures from image files stored on disk using the `D3DXCreateTextureFromFile` function.
- We can filter textures by using the minification, magnification, and mipmap filter sampler states.
- Address modes define what Direct3D is supposed to do with texture coordinates outside the [0, 1] range. For example, should the texture be tiled, mirrored, clamped, etc.?

7. Blending

Blending is a technique that allows us to combine pixels to previously drawn primitives. This way, we can achieve various effects, like transparency.

Suppose we want to draw the scene from in figure 7.1, which shows a crate and a transparent teapot in front of it. We need to combine the pixel colors of the teapot with the pixel colors of the crate. This can be achieved by combining the pixels of the primitive currently being rasterized with the pixel values previously written at the same locations on the back buffer. This idea is called blending.
Keywords: blending, `D3DRS_ALPHABLENDENABLE`, transparency, alpha channel, DDS file, DirectX Texture tool

Figure 7.1: Transparent teapot

When using blending, the following rule should be followed: "Draw objects that do not use blending first. Then sort the objects that use blending by their distance from the camera; this is most efficiently done if the objects are in view space so that you can sort simply by the z-component. Finally, draw the objects that use blending in a back-to-front order."

The blending of two pixel values is calculated as follows:
*OutputPixel = SourcePixel * SourceBlendFactor + DestPixel * DestBlendFactor*

- *OutputPixel*: The resulting blended pixel
- *SourcePixel*: The pixel that is to be blended with the pixel on the back buffer
- *DestPixel*: The pixel on the back buffer
- *SourceBlendFactor* and *DestBlendFactor*: A value between 0 and 1, specifies the percent of the pixel to use in the blend

Blending is disabled by default, because it is not a cheap operation. You can enable it by setting the `D3DRS_ALPHABLENDENABLE` render state to true:

```
Device->SetRenderState(D3DRS_ALPHABLENDENABLE, true);
```

Remember to only enable it for geometry that needs it. When done rendering that geometry, disable alpha blending.

By setting different combinations of source and destination blend factors, you can create dozens of different blending effects. You can set the source blend factor and the destination blend factor by setting the `D3DRS_SRCBLEND` and `D3DRS_DESTBLEND` render states, respectively.

```
Device->SetRenderState(D3DRS_SRCBLEND, Source);
Device->SetRenderState(D3DRS_DESTBLEND, Destination);
```

`Source` and `Destination` can be one of the values of the `D3DBLEND` Enumerated Type. This enumeration can be found in the SDK documentation for a full list of its values. Examples of some of these values:

- `D3DBLEND_ZERO`: Blend factor is (0, 0, 0, 0).
- `D3DBLEND_ONE`: Blend factor is (1, 1, 1, 1).
- `D3DBLEND_SRCALPHA`: Blend factor is (A_s, A_s, A_s, A_s). This is the default source blend factor.
- `D3DBLEND_INVSRCALPHA`: Blend factor is (1 - A_s, 1 - A_s, 1 - A_s, 1 - A_s). This is the default destination blend factor.

7.2 Transparency

The alpha component of a vertex color was ignored in previous chapter. It is primarily used in blending. It is mainly for specifying the level of transparency of a pixel. If there were 8 bits reserved for the alpha component, its range would be [0, 255], which corresponds to [0%, 100%] opacity.
Set the source and destination blend factor to respectively `D3DBLEND_SRCALPHA` and `D3DBLEND_INVSRCALPHA` to make the alpha component describe the transparency level of the pixels. These values are the default blend factors.

7.2.1 Alpha Channels

We can obtain alpha information from two sources, alpha components or a texture's *alpha channel*. The latter is an extra set of bits reserved for each texel that stores an alpha component. When a texture is mapped, the alpha component from the alpha channel is mapped also. Figure 7.2 shows an 8-bit grayscale map representing the alpha channel of a texture.

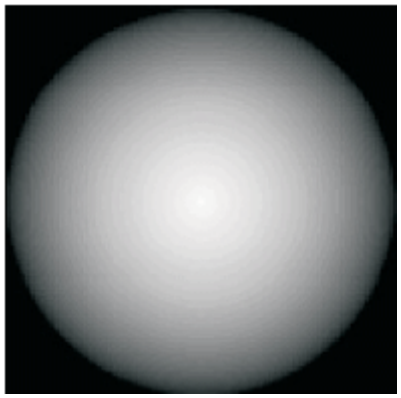


Figure 7.2

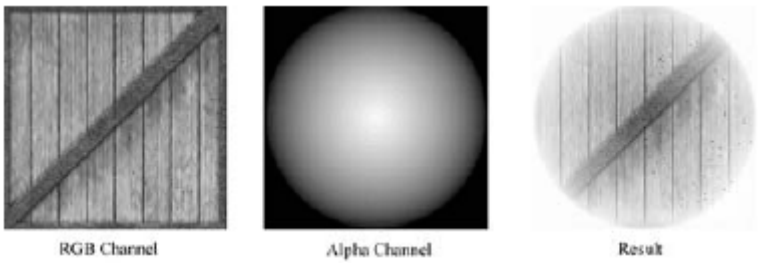


Figure 7.3

Figure 7.3 shows the result of rendering a textured quad with an alpha channel specifying the parts that are transparent.

If the currently set texture:

- has an alpha channel, the alpha is taken from the alpha channel.
- has no alpha channel, the alpha is obtained from the vertex color.

You can manually specify which source to use:

```
// compute alpha from diffuse colors during shading
Device->SetTextureStageState(0, D3DTSS_ALPHAARG1, D3DTA_DIFFUSE);
```

```
Device->SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1);

// take alpha from alpha channel
Device->SetTextureStageState(0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE);
Device->SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1);
```

7.2.2 Creating an Alpha Channel using the DirectX Texture Tool

In this section we will see how to create a DDS file using the DirectX Texture Tool. A DDS file is an image file designed for DirectX applications and textures.
We will use the image files from the texAlpha sample. Open the crate.jpg file with the Texture Tool. The crate is automatically loaded in as a 24-bit RGB texture with 8 bits of red, 8 bits of green, and 8 bits of blue per pixel. We need 8-bits extra for the alpha channel, so we change it to a 32-bit ARGB texture. Select **Format->Change Surface Format**. In the dialog box that pops up, select ”Unsigned 32-bit: A8R8G8B8” from the drop down list. The crate image will then be changed to 32-bit. See figure 7.4.

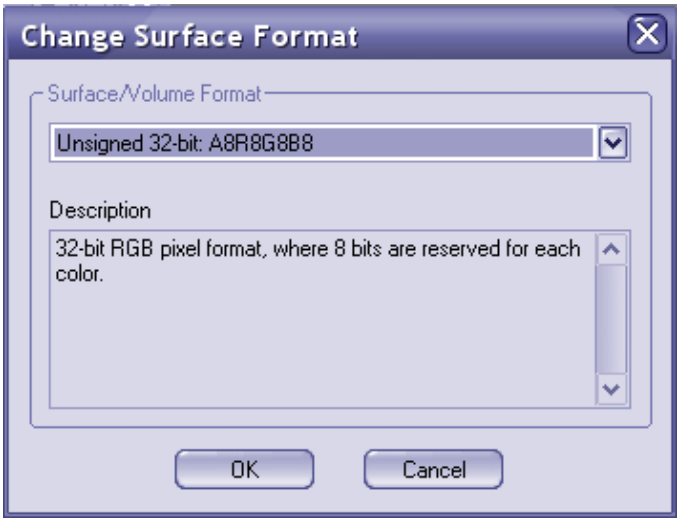


Figure 7.4: Changing the format

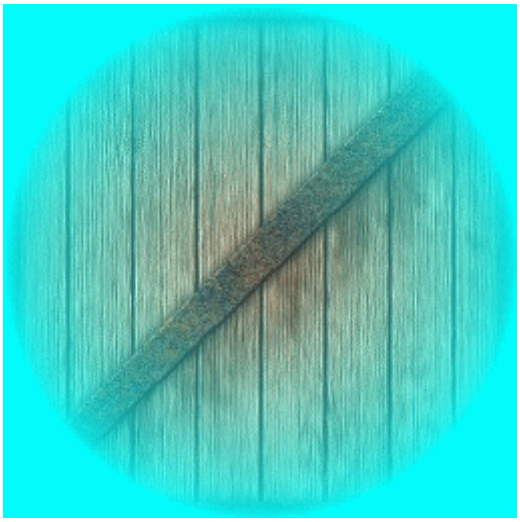


Figure 7.5: Resulting texture with a alpha channel

Next, select **File** from the menu, and then choose **Open Onto Alpha Channel Of This Texture**. Locate the alphachannel.bmp file. Figure 7.5 shows the result after the alpha channel data has been inserted.

7.3 Sample: Transparency

The sample application located in the MtrlAlpha folder draws the scene as shown in figure 7.1. This sample allows you to increase/decrease the alpha component with the A and S keys.
Steps required when using blending:
Step 1. Set the blend factors `D3DRS_SRCBLEND` and `D3DRS_DESTBLEND`.
Step 2. If using the alpha component, specify the source (material or alpha channel).
Step 3. Enable the alpha blending render state.

As usual, I will show the steps in the code comments.
The following global variables are used:

```
ID3DXMesh* Teapot = 0; // the teapot
D3DMATERIAL9 TeapotMtrl; // the teapot's material
IDirect3DVertexBuffer9* BkGndQuad = 0; // background quad - crate
IDirect3DTexture9* BkGndTex = 0; // crate texture
```

```
D3DMATERIAL9 BkGndMtrl; // background material
```

Shown below is the `Setup` method with much omitted code. In this example we instruct the alpha to be taken from the diffuse component of the material. Also, the diffuse alpha component is set to 0.5, meaning 50% transparency.

```
bool Setup() {
      TeapotMtrl = d3d::RED_MTRL;
      TeapotMtrl.Diffuse.a = 0.5f; // set alpha to 50% opacity
      BkGndMtrl = d3d::WHITE_MTRL;

      D3DXCreateTeapot(Device, &Teapot, 0);

      ...// Create background quad snipped
      ...// Light and texture setup snipped

      // use alpha in material's diffuse component for alpha
      Device->SetTextureStageState(0, D3DTSS_ALPHAARG1,
                                      D3DTA_DIFFUSE);
      Device->SetTextureStageState(0, D3DTSS_ALPHAOP,
                                      D3DTOP_SELECTARG1);

      // STEP 1: set blending factors so that alpha
      // component determines transparency
      Device->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
      Device->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);

      ...// view/projection matrix setup snipped

      return true;
}
```

In the `Display` function, we check to see if the A or S key was pressed and respond by increasing or decreasing the material's alpha value. We then render the background quad. Finally, we enable alpha blending, render the teapot with alpha blending enabled, and then disable alpha blending.

```
bool Display(float timeDelta) {
      if( Device ) {
            // increase/decrease alpha via keyboard input
            if( ::GetAsyncKeyState('A') & 0x8000f )
                  TeapotMtrl.Diffuse.a += 0.01f;
            if( ::GetAsyncKeyState('S') & 0x8000f )
                  TeapotMtrl.Diffuse.a -= 0.01f;
            // force alpha to [0, 1] interval
            if(TeapotMtrl.Diffuse.a > 1.0f)
                  TeapotMtrl.Diffuse.a = 1.0f;
            if(TeapotMtrl.Diffuse.a < 0.0f)
                  TeapotMtrl.Diffuse.a = 0.0f;

            // Render
            Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
            0xffffffff, 1.0f, 0);
            Device->BeginScene();
            // Draw the background
            D3DXMATRIX W;
            D3DXMatrixIdentity(&W);
            Device->SetTransform(D3DTS_WORLD, &W);
            Device->SetFVF(Vertex::FVF);
            Device->SetStreamSource(0, BkGndQuad, 0, sizeof(Vertex));
```

```
            Device->SetMaterial(&BkGndMtrl);
            Device->SetTexture(0, BkGndTex);
            Device->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 2);
            // Draw the teapot
            // STEP 3: Enable the alpha blending state
            Device->SetRenderState(D3DRS_ALPHABLENDENABLE, true);
            D3DXMatrixScaling(&W, 1.5f, 1.5f, 1.5f);
            Device->SetTransform(D3DTS_WORLD, &W);
            Device->SetMaterial(&TeapotMtrl);
            Device->SetTexture(0, 0);
            Teapot->DrawSubset(0);
            Device->SetRenderState(D3DRS_ALPHABLENDENABLE, false);
            Device->EndScene();
            Device->Present(0, 0, 0, 0);
      }
      return true;
}
```

7.4 Summary

- Alpha blending allows combining the pixels of the primitive currently being rasterized with the pixel values previously written at the same locations on the back buffer.
- The blend factors allow us to control how the source and destination pixels are blended together.
- Alpha information can come from the diffuse component of the primitive's material alpha channel of the primitive's texture.

8. Fonts

In this chapter shows how to display some text to the user. This will be achieved by using the `ID3DXFont` interface, `CD3DFont` class or the `D3DXCreateText` function.
Keywords: `ID3DXFont, CD3DFont, D3DXCreateText, DrawText`

8.1 ID3DXFont

The `ID3DXFont` interface can handle complex fonts because it uses GDI (Graphical Device Interface, see http://www.webopedia.com/TERM/G/GDI.html). It can be created with the `D3DXCreateFontIndirect` function:

```
HRESULT WINAPI D3DXCreateFontIndirect(
      LPDIRECT3DDEVICE9 pDevice,
      CONST D3DXFONT_DESC *pDesc,
      LPD3DXFONT *ppFont
);
```

Prior to the DirectX 9.0b update, this function used a `LOGFONT` struct as the second parameter. The `LOGFONT` and `D3DXFONT_DESC` structs have almost the same members. The samples and the book still use the `LOGFONT` struct, but the code shown here uses `D3DXFONT_DESC`. To change the samples from the book (to make it work/compile with the latest SDK versions), just replace `LOGFONT` with `D3DXFONT_DESC` and remove the members that `LOGFONT` has which `D3DXFONT_DESC` does not have.
Example usage of `D3DXCreateFontIndirect`:

```
D3DXFONT_DESC fontDesc;
ZeroMemory(&fontDesc, sizeof(D3DXFONT_DESC));

fontDesc.Height      = 25;
fontDesc.Width = 12;
```

```
fontDesc.Weight     = 500;
fontDesc.MipLevels  = D3DX DEFAULT;
fontDesc.Italic     = false;
fontDesc.CharSet    = 0;
fontDesc.OutputPrecision = 0;
fontDesc.Quality    = 0;
fontDesc.PitchAndFamily  = 0;
strcpy(fontDesc.FaceName, "Times New Roman"); // font style

ID3DXFont* Font   = 0;
D3DXCreateFontIndirect(Device, &fontDesc, &Font);
```

The `D3DXFONT_DESC` structure must be filled to describe the font we want to create.

8.1.1 Drawing with ID3DXFont

To draw some text, simply call the `ID3DXFont::DrawText` method.

```
INT DrawText(
      LPD3DXSPRITE pSprite,
      LPCTSTR pString,
      INT Count,
      LPRECT pRect,
      DWORD Format,
      D3DCOLOR Color
);
```

- `pSprite`: Pointer to an ID3DXSprite object that contains the string. This chapter does not use this object. It can be set to NULL.
- `pString`: Pointer to the string to draw
- `Count`: Number of characters in the string. We can specify –1 if the string is null terminating.
- `pRect`: Pointer to a `RECT` structure that defines the area on the screen to which the text is to be drawn
- `Format`: Optional flags that specify how the text should be formatted; see the SDK documentation for details.
- `Color`: The text color (see `D3DCOLOR`)

Example usage:

```
Font->DrawText(
      NULL,
      "Hello World", // String to draw.
      -1, // Null terminating string.
      &rect, // Rectangle to draw the string in.
      DT_TOP | DT_LEFT, // Draw in top-left corner of rect.
      0xff000000); // Black.
```

8.2 CD3DFont

`CD3DFont` uses Direct3D for rendering instead of GDI, making it much faster that `ID3DXFont`. However, `CD3DFont` does not support complex fonts. Use `CD3DFont` when you only need a simple font.
The following files are needed for `CD3DFont`: d3dfont.h, d3dfont.cpp, d3dutil.h, d3dutil.cpp, dxutil.h, and dxutil.cpp. They can be found at the CFont sample directory.

To create a `CD3DFont` instance, simply call its constructor:

```
CD3DFont(const TCHAR* strFontName,DWORD dwHeight,DWORD dwFlags=0L);
```

- `strFontName`: A null-terminated string that specifies the typeface name of the font
- `dwHeight`: The height of the font
- `dwFlags`: Optional creation flags; you can set this parameter to zero or use a combination of the following flags; `D3DFONT_BOLD, D3DFONT_ITALIC, D3DFONT_ZENABLE`.

After we have instantiated a `CD3DFont` object, we must call the following methods (in the order shown) that initialize the font:

```
Font = new CD3DFont("Times New Roman", 16, 0); // instantiate
Font->InitDeviceObjects( Device );
Font->RestoreDeviceObjects();
```

8.2.1 Drawing with CD3DFont

`CD3DFont` also has a `DrawText` method for drawing text:

```
HRESULT CD3DFont::DrawText(FLOAT x, FLOAT y, DWORD dwColor,
                      const TCHAR* strText, DWORD dwFlags=0L);
```

- `x`: The x-coordinate in screen space of where to begin drawing the text
- `y`: The y-coordinate in screen space of where to begin drawing the text
- `dwColor`: The color of the text
- `strText`: Pointer to the string to draw
- `dwFlags`: Optional rendering flags; you can set this parameter to 0 or use a combination of the following: `D3DFONT_CENTERED, D3DFONT_TWOSIDED, D3DFONT_FILTERED`.

8.2.2 Cleanup

Before deleting a `CD3DFont` object, call some cleanup routines first:

```
Font->InvalidateDeviceObjects();
Font->DeleteDeviceObjects();
delete Font;
```

8.3 D3DXCreateText

`D3DXCreateText` creates a 3D mesh containing the specified text.

```
HRESULT WINAPI D3DXCreateText(
      LPDIRECT3DDEVICE9 pDevice,
      HDC hDC,
      LPCTSTR pText,
      FLOAT Deviation,
      FLOAT Extrusion,
      LPD3DXMESH *ppMesh,
      LPD3DXBUFFER *ppAdjacency,
      LPGLYPHMETRICSFLOAT pGlyphMetrics
);
```

- `pDevice`: The device to be associated with the mesh

- `hDC`: A handle to a device context that contains a description of the font that we are going to use to generate the mesh
- `pText`: Pointer to a null-terminating string specifying the text to create a mesh of
- `Deviation`: Maximum chordal deviation from TrueType font outlines. This value must be greater than or equal to 0. When this value is 0, the chordal deviation is equal to one design unit of the original font.
- `Extrusion`: The depth of the font measured in the negative z-axis direction
- `ppMesh`: Returns the created mesh
- `ppAdjacency`: Returns the created mesh's adjacency info. Specify null if you don't need this.
- `pGlyphMetrics`: A pointer to an array of `LPGLYPHMETRICSFLOAT` structures that contain the glyph metric data. You can set this to 0 if you are not concerned with glyph metric data.

To draw some text looking like this:

Figure 8.1: 3D text created with the D3DXCreateText function

Use the following code:

```
// Obtain a handle to a device context.
HDC hdc = CreateCompatibleDC( 0 );

//Fill out a LOGFONT structure that describes the font's properties.
LOGFONT lf;
ZeroMemory(&lf, sizeof(LOGFONT));
lf.lfHeight = 25; // in logical units
lf.lfWidth = 12; // in logical units
lf.lfWeight = 500; // boldness, range 0(light) - 1000(bold)
lf.lfItalic = false;
lf.lfUnderline = false;
lf.lfStrikeOut = false;
lf.lfCharSet = DEFAULT_CHARSET;
strcpy(lf.lfFaceName, "Times New Roman"); // font style

// Create a font and select that font with the device context.
HFONT hFont;
HFONT hFontOld;
hFont = CreateFontIndirect(&lf);
hFontOld = (HFONT)SelectObject(hdc, hFont);

// Create the 3D mesh of text.
ID3DXMesh* Text = 0;
D3DXCreateText(_device, hdc, "Direct3D", 0.001f, 0.4f, &Text, 0, 0);

// Reselect the old font, and free resources.
SelectObject(hdc, hFontOld);
DeleteObject( hFont );
DeleteDC( hdc );
```

The drawing of the 3D text mesh can be done by:

```
Text->DrawSubset( 0 );
```

8.4 Font Samples

The font samples are located in Book Part III Code\Chapter 9. The ID3DXFont sample will not compile with the latest DirectX 9 SDK because of the `LOGFONT` problem (see section 8.1). A modified version can be found at:
http://www.cs.vu.nl/~tljchung/directx/samples/ID3DXFont_adjusted_sample.zip.

8.5 Summary

- Use the `ID3DXFont` interface to render text when you need to support complex fonts and formatting. This interface uses GDI internally to render text and is therefore slower.
- Use `CD3DFont` to render simple text quickly. This class uses textured triangles and Direct3D to render text and is therefore much faster than `ID3DXFont`.
- Use `D3DXCreateText` to create a 3D mesh of a string of text.

9. Stenciling

Direct3D supports an additional buffer called the stencil buffer. It is an off-screen buffer and has the same resolution as the back buffer and depth buffer. The stencil buffer allows us to block rendering certain parts of the back buffer. This means your software can "mask" portions of the rendered image so that they aren't displayed.
For an example see figure 9.1. The screenshot shows the implementation of a mirror. We use the stencil buffer to block rendering when it is not on the mirror.

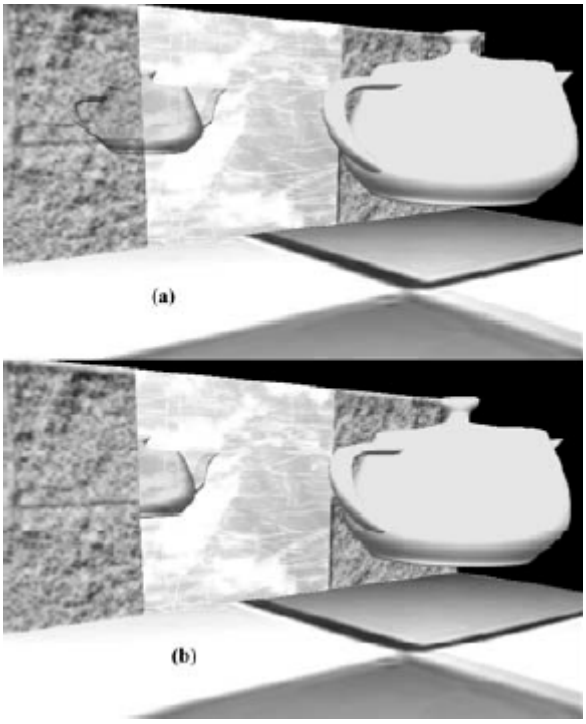


Figure 9.1a: Teapot reflected without a stencil buffer.

Figure 9.1b: Teapot reflected with a stencil buffer.

This chapter uses an example to explain stenciling. This example implements the mirror shown in figure 9.1. Stenciling is a rather complicated subject. I recommend reading about it in Luna's book (part 2, chapter 8) and in the SDK documentation.

Keywords: stencil buffer, D3DRS_STENCILENABLE, stencil test, ref, mask, value, comparison operation, stencil write mask, mirror, reflection matrix, D3DXMatrixReflect, planar shadow, shadow matrix, double blending

9.1 Using the stencil buffer

We can enable a stencil buffer with the following code:

```
Device->SetRenderState(D3DRS_STENCILENABLE, true);

... // do stencil work

// disable the stencil buffer
Device->SetRenderState(D3DRS_STENCILENABLE, false);
```

The IDirect3DDevice9::Clear method is used to clear the stencil buffer to a default value.

```
Device->Clear(0, 0,
           D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER | D3DCLEAR_STENCIL,
           0xff000000, 1.0f, 0 );
```

The flags given in the third argument indicate that the back buffer (D3DCLEAR_TARGET), depth buffer (D3DCLEAR_ZBUFFER) and stencil buffer (D3DCLEAR_STENCIL) should be cleared.

9.1.1 Requesting a Stencil Buffer

The stencil and depth buffer share the same off-screen surface buffer. To request a stencil buffer, specify the format of the stencil buffer at the same time as when specifying the format of the depth buffer. A segment of memory in each pixel is designated to each particular buffer. The following depth/stencil formats illustrate this:

- D3DFMT_D24S8: This format says to create a 32-bit depth/stencil buffer and designate 24 bits per pixel to the depth buffer and 8 bits per pixel to the stencil buffer.
- D3DFMT_D24X4S4: 32-bit depth/stencil buffer, 24 bits per pixel to the depth buffer, 4 bits per pixel to the stencil buffer. Four of the bits will not be used.

9.1.2 Stencil Test

As previously stated, we can use the stencil buffer to block rendering to certain areas of the back buffer. To determine whether a pixel should be written is given by the following expression:

```
(ref & mask) ComparisonOperation (value & mask)
```

The variables are:

- *ref*: stencil reference value
- *mask*: application defined masking value
- *value*: the particular pixel that we are testing

How the expression works:

1. Perform a bitwise AND operation on *ref* with *mask*.
2. Perform a bitwise AND operation on *value* with *mask*.

3. Compare the results of Step 1 (this will called A) and Step 2 (will be called B) by using the comparison operation.

A Boolean value will be returned as result. If this value is

- true: write the pixel to the back buffer.
- false: block the pixel from being written to the back buffer. If a pixel isn't written to the back buffer, it isn't written to the depth buffer either.

9.1.3 Controlling the Stencil Test

In this section we will see how to control the variables used in the stencil test.

Stencil Reference Value (ref)

The *ref* value is zero by default, but we can change it with the `D3DRS_STENCILREF` render state. To set the stencil reference value to one:

```
Device->SetRenderState(D3DRS_STENCILREF, 0x1);
```

We tend to use hexadecimal because it makes it easier to see the bit alignment of an integer, and this is useful to see when doing bit-wise operations, such as the AND operation.

Stencil Mask (mask)

The *mask* value is used to mask (hide) bits in both the *ref* and *value* variables. The default mask is 0xffffffff, which doesn't mask any bits. We can change the mask by setting the `D3DRS_STENCILMASK` render state. The following example masks the 16 high bits:

```
Device->SetRenderState(D3DRS_STENCILMASK, 0x0000ffff);
```

Stencil Value (value)

This is the value in the stencil buffer for the current pixel that we are stencil testing. If we are performing the stencil test on the ij^{th} pixel, then value will be the value in the ij^{th} entry of the stencil buffer.
We can not explicitly set individual stencil values, but we can clear the stencil buffer. In addition we can use the stencil render states to control what gets written to the stencil buffer. These render states will be covered shortly.

Comparison Operation

We can set the comparison operation by setting the `D3DRS_STENCILFUNC` render state. The comparison operation can be any member of the `D3DCMPFUNC` enumerated type:

```
typedef enum _D3DCMPFUNC {
      D3DCMP_NEVER = 1,
      D3DCMP_LESS = 2,
      D3DCMP_EQUAL = 3,
      D3DCMP_LESSEQUAL = 4,
      D3DCMP_GREATER = 5,
      D3DCMP_NOTEQUAL = 6,
      D3DCMP_GREATEREQUAL = 7,
      D3DCMP_ALWAYS = 8,
      D3DCMP_FORCE_DWORD = 0x7fffffff
} D3DCMPFUNC;
```

- `D3DCMP_NEVER`: Always fail the test.

- D3DCMP_LESS: Accept the new pixel if its value is less than the value of the current pixel. A < B, where A means (ref & mask) and B means (value & mask). See section 9.1.2.
- D3DCMP_EQUAL: Accept the new pixel if its value equals the value of the current pixel. A = B
- D3DCMP_LESSEQUAL: Accept the new pixel if its value is less than or equal to the value of the current pixel. A ≤ B
- D3DCMP_GREATER: Accept the new pixel if its value is greater than the value of the current pixel. A > B
- D3DCMP_NOTEQUAL: Accept the new pixel if its value does not equal the value of the current pixel. A ≠ B
- D3DCMP_GREATEREQUAL: Accept the new pixel if its value is greater than or equal to the value of the current pixel. A ≥ B
- D3DCMP_ALWAYS: Always pass the test.
- D3DCMP_FORCE_DWORD: Forces this enumeration to compile to 32 bits in size. Without this value, some compilers would allow this enumeration to compile to a size other than 32 bits. This value is not used.

9.1.3 Updating the Stencil Buffer

In addition to deciding whether to write or block a particular pixel from being written to the back buffer, we can define how the stencil buffer entry should be updated based on three possible cases which are represented by render states:

- D3DRENDERSTATE_STENCILFAIL: Indicates the stencil operation to perform if the stencil test fails. The stencil operation can be one of the members of the D3DSTENCILOP enumerated type. The default value is D3DSTENCILOP_KEEP.
- D3DRENDERSTATE_STENCILZFAIL: Indicates the stencil operation to perform if the stencil test passes and the depth test fails. The operation can be one of the members of the D3DSTENCILOP enumerated type. The default value is D3DSTENCILOP_KEEP.
- D3DRENDERSTATE_STENCILPASS: Indicates the stencil operation to perform if both the stencil test and the depth test pass. The operation can be one of the members of the D3DSTENCILOP enumerated type. The default value is D3DSTENCILOP_KEEP.

The D3DSTENCILOP enumerated type describes the stencil operations for the these three render states:

```
typedef enum _D3DSTENCILOP {
      D3DSTENCILOP_KEEP = 1,
      D3DSTENCILOP_ZERO = 2,
      D3DSTENCILOP_REPLACE = 3,
      D3DSTENCILOP_INCRSAT = 4,
      D3DSTENCILOP_DECRSAT = 5,
      D3DSTENCILOP_INVERT = 6,
      D3DSTENCILOP_INCR = 7,
      D3DSTENCILOP_DECR = 8,
      D3DSTENCILOP_FORCE_DWORD = 0x7fffffff
} D3DSTENCILOP;
```

- D3DSTENCILOP_KEEP: Specifies to not change the stencil buffer (that is, keep the value currently there)
- D3DSTENCILOP_ZERO: Specifies to set the stencil buffer entry to zero

- `D3DSTENCILOP_REPLACE`: Specifies to replace the stencil buffer entry with the stencil reference value
- `D3DSTENCILOP_INCRSAT`: Specifies to increment the stencil buffer entry. If the incremented value exceeds the maximum allowed value, we clamp the entry to that maximum.
- `D3DSTENCILOP_DECRSAT`: Specifies to decrement the stencil buffer entry. If the decremented value is less than zero, we clamp the entry to zero.
- `D3DSTENCILOP_INVERT`: Specifies to invert the bits of the stencil buffer entry
- `D3DSTENCILOP_INCR`: Specifies to increment the stencil buffer entry. If the incremented value exceeds the maximum allowed value, we wrap to zero.
- `D3DSTENCILOP_DECR`: Specifies to decrement the stencil buffer entry. If the decremented value is less than zero, we wrap to the maximum allowed value.
- `D3DSTENCILOP_FORCE_DWORD`: Forces this enumeration to be compiled to 32 bits. Without this value, some compilers would allow this enumeration to compile to a size other than 32 bits. This value isn't used.

Suppose we want to set the stencil buffer entry to zero when the stencil test fails, we would write the following code:

```
Device->SetRenderState(D3DRS_STENCILFAIL, D3DSTENCILOP_ZERO);
```

9.1.4 Stencil Write Mask

In addition to the mentioned stencil render states, we can set a write mask that will mask off bits of any value that we write to the stencil buffer. We can set the write mask with the `D3DRS_STENCILWRITEMASK` render state. The default value is 0xffffffff. The following example masks the top 16 bits:

```
Device->SetRenderState(D3DRS_STENCILWRITEMASK, 0x0000ffff);
```

Don't confuse this with `D3DRENDERSTATE_STENCILMASK`. This render state specifies the mask to apply to the reference value and each stencil buffer entry (see 9.1.3).

9.2 Sample: Mirror

This part describes how to implement a mirror. The sample can be found at part 2 chapter 8, "Stencil Mirror" folder. For simplicity, the mirror is only implemented for planar surfaces. Like a mirror hanging on a wall or a shiny floor.
Implementing a mirror requires solving two problems:

1. Reflect an object about an arbitrary plane, see section 9.2.1
2. Only display the reflection in the mirror (see figure 9.1b), we will use the stencil buffer to achieve this

9.2.1 Reflecting about an arbitrary plane

I am not going to explain the mathematics for achieving this effect. We will simply use the matrix **R** to transform a point **v** to its reflected point **v'**.

$$\mathbf{R} = \begin{bmatrix} -2n_x n_x + 1 & -2n_y n_x & -2n_z n_x & 0 \\ -2n_x n_y & -2n_y n_y + 1 & -2n_z n_y & 0 \\ -2n_x n_z & -2n_y n_z & -2n_z n_z + 1 & 0 \\ -2n_x d & -2n_y d & -2n_z d & 1 \end{bmatrix}$$

This matrix can also be found at the SDK documentation. Search for "D3DXMatrixReflect" in the index. Use the D3DX library function `D3DXMatrixReflect` to generate this reflection matrix for a given plane.

```
D3DXMATRIX *WINAPI D3DXMatrixReflect(
      D3DXMATRIX *pOut, // The resulting reflection matrix
      CONST D3DXPLANE *pPlane // The plane to reflect about
);
```

To get an arbitrary plane, the function `D3DXPlaneFromPointNormal` can be used. This function returns the plane using a point (which is on the plane) and the plane's normal.

```
D3DXPLANE *WINAPI D3DXPlaneFromPointNormal(
      D3DXPLANE *pOut, // The resulting plane
      CONST D3DXVECTOR3 *pPoint, // point on the plane
      CONST D3DXVECTOR3 *pNormal // plane's normal
);
```

The following code shows how to get the xy plane.

```
D3DXPLANE plane; // the resulting plane
D3DXVECTOR3 point(0.0f, 0.0f, 0.0f); // a point on the xy plane
D3DXVECTOR3 normal(0.0f, 0.0f, 1.0f); // xy plane's normal

D3DXPlaneFromPointNormal(&plane, &pnt, &nrm);
```

There are three special case reflection transformations. They are the reflections about the three standard coordinate planes, the yz plane, xz plane, and xy plane, and are represented by the following three matrices, respectively:

$$\mathbf{R}_{yz} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R}_{xz} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R}_{xy} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To reflect a point across the yz plane, we simply take the opposite of the z-component. Similarly, to reflect a point across the xz plane, we take the opposite of the y-component. Finally, to reflect a point across the xy plane, we take the opposite of the z-component. These reflections are readily seen by observing the symmetry on each of the standard coordinate planes.

9.2.2 Mirror Implementation

When implementing a mirror, an object is only reflected if it is in front of a mirror. To simplify things, we always reflect the object and render it no matter where it is. As said before, to avoid the problem seen in figure 9.1a, we will use the stencil buffer.
The following steps briefly outline how the mirror application is implemented:

1. Render the entire scene as normal, the floor, walls, mirror, and teapot, but **not** the teapot's reflection.
2. Clear the stencil buffer to 0.
3. Render the primitives that make up the mirror into the stencil buffer only. Set the stencil test to always succeed, and specify that the stencil buffer entry should be replaced with 1 if the test passes. Since we are only rendering the mirror, all the pixels in the stencil buffer will be 0 except for the pixels that correspond to the mirror, they will have a 1. Essentially, we are marking the pixels of the mirror in the stencil buffer.

4. Now we render the reflected teapot to the back buffer and stencil buffer. But recall that we only render to the back buffer if the stencil test passes. This time we set the stencil test to only succeed if the value in the stencil buffer is a 1. In this way, the teapot is only rendered to areas that have a 1 in their corresponding stencil buffer entry. Since the areas in the stencil buffer that correspond to the mirror are the only entries that have a 1, the reflected teapot is only rendered into the mirror.

9.2.3 Sample Application's code

The code relevant to this sample lies in the `RenderMirror` function, which first renders the mirror primitives to the stencil buffer and then renders the reflected teapot only if it is being rendered into the mirror. We walk through the `RenderMirror` function almost line by line and explain what is occurring and, more importantly, why.
If you are using the steps outlined in section 9.2.2 to serve as an overall guide to the code, note that we are starting at step 3 since steps 1 and 2 have nothing to do with the stencil buffer. Also be aware that we are discussing the rendering of the mirror through this explanation.

Enabling the stencil buffer

```
void RenderMirror()
{
Device->SetRenderState(D3DRS_STENCILENABLE, true);
Device->SetRenderState(D3DRS_STENCILFUNC, D3DCMP_ALWAYS);
Device->SetRenderState(D3DRS_STENCILREF, 0x1);
Device->SetRenderState(D3DRS_STENCILMASK, 0xffffffff);
Device->SetRenderState(D3DRS_STENCILWRITEMASK,0xffffffff);
Device->SetRenderState(D3DRS_STENCILZFAIL, D3DSTENCILOP_KEEP);
Device->SetRenderState(D3DRS_STENCILFAIL, D3DSTENCILOP_KEEP);
Device->SetRenderState(D3DRS_STENCILPASS, D3DSTENCILOP_REPLACE);
```

We set the stencil comparison operation to `D3DCMP_ALWAYS`, specifying that the stencil test will always pass.
If the depth test fails, we specify `D3DSTENCILOP_KEEP`, which indicates to not update the stencil buffer entry. That is, we keep its current value. We do this because if the depth test fails, it means the pixel is obscured. Therefore, we do not want to render part of the reflection to a pixel that is obscured.
We also specify `D3DSTENCILOP_KEEP` if the stencil test fails. But this isn't really necessary here, since the test never fails because we specified `D3DCMP_ALWAYS`. However, we change the comparison operation in just a bit, so setting the stencil fail render state is required eventually; we just do it now.
If the depth and stencil tests pass, we specify `D3DSTENCILOP_ REPLACE`, which replaces the stencil buffer entry with the stencil reference value, 0x1.

Render the mirror to the stencil buffer

This next block of code renders the mirror, but only to the stencil buffer. We can stop writes to the depth buffer by setting the `D3DRS_ZWRITEENABLE` and specifying false. We can prevent updating the back buffer with blending and setting the source blend factor to `D3DBLEND_ZERO` and the destination blend factor to `D3DBLEND_ONE`. Plugging these blend factors into the blending equation, we show that the back buffer is left unchanged[ii]:

[ii] Note that in the equation, the * operator means component-wise multiplication. Example:
$(a_1, a_2, a_3) * (b_1, b_2, b_3) = (a_1b_1, a_2b_2, a_3b_3)$

FinalPixel = *sourcePixel * (0, 0, 0, 0) + DestPixel * (1, 1, 1, 1)*
= *(0, 0, 0, 0) + DestPixel*
= *DestPixel*

```
// disable writes to the depth and back buffers
Device->SetRenderState(D3DRS_ZWRITEENABLE, false);
Device->SetRenderState(D3DRS_ALPHABLENDENABLE, true);
Device->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ZERO);
Device->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);

// draw the mirror to the stencil buffer
Device->SetStreamSource(0, VB, 0, sizeof(Vertex));
Device->SetFVF(Vertex::FVF);
Device->SetMaterial(&MirrorMtrl);
Device->SetTexture(0, MirrorTex);
D3DXMATRIX I;
D3DXMatrixIdentity(&I);
Device->SetTransform(D3DTS_WORLD, &I);
Device->DrawPrimitive(D3DPT_TRIANGLELIST, 18, 2);

// re-enable depth writes
Device->SetRenderState(D3DRS_ZWRITEENABLE, true);
```

At this point, the pixels in the stencil buffer that correspond to the visible pixels of the mirror have an entry on 0x1, thus marking the area where the mirror has been rendered. We now prepare to render the reflected teapot. We only want to render the reflection into pixels that correspond to the mirror. This can be easily done now that we have marked those pixels in the stencil buffer.
We set the following render states:

```
Device->SetRenderState(D3DRS_STENCILFUNC, D3DCMP_EQUAL);
Device->SetRenderState(D3DRS_STENCILPASS, D3DSTENCILOP_KEEP);
```

Thus making the stencil test:

```
      (ref & mask)     ==    (value & mask)
(0x1 & 0xffffffff)     ==    (value & 0xffffffff)
           (0x1)       ==    (value & 0xffffffff)
```

This shows that the stencil test only succeeds if *value = 0x1*. Since *value* is only *0x1* in areas of the stencil buffer that correspond to the mirror, the test only succeeds if we are rendering to those areas. Thus, the reflected teapot is only drawn into the mirror and is not drawn into other surfaces.
We have changed the `D3DRS_STENCILPASS` render state to `D3DSTENCILOP_KEEP`, which means keeping the value in the stencil buffer if the test passed. Therefore, in this next rendering pass, we do not change the values in the stencil buffer (all controls are `D3DSTENCILOP_KEEP`). We are only using the stencil buffer to mark the pixels that correspond to the mirror.

Compute the reflection matrix

The next part of the `RenderMirror` function computes the matrix that positions the reflection in the scene:

```
// position reflection
D3DXMATRIX W, T, R;
D3DXPLANE plane(0.0f, 0.0f, 1.0f, 0.0f); // xy plane
```

```
D3DXMatrixReflect(&R, &plane);

D3DXMatrixTranslation(&T,
      TeapotPosition.x,
      TeapotPosition.y,
      TeapotPosition.z);

W = T * R;
```

We first translated to the non-reflection teapot position. Then it is reflected across the xy plane. This plane is one of the special planes described in section 9.2.1, that is why we could simply use (0.0f, 0.0f, 1.0f, 0.0f) as the plane.

Render the reflected teapot

If we render the reflected teapot now, it will not be displayed. Because the reflected teapot's depth is greater than the mirror's depth, and thus the mirror primitives technically obscure the reflected teapot. To get around this, we clear the depth buffer:

```
Device->Clear(0, 0, D3DCLEAR_ZBUFFER, 0, 1.0f, 0);
```

However, if we simply clear the depth buffer, the reflected teapot is drawn in front of the mirror and things do not look right. What we want to do is clear the depth buffer in addition to blending the reflected teapot with the mirror. In this way, the reflected teapot looks like it is "in" the mirror. This can be done with the following equation:

FinalPixel = *sourcePixel * destPixel + DestPixel * (0, 0, 0, 0)*
= *sourcePixel * destPixel*

In code this would be:

```
Device->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_DESTCOLOR);
Device->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ZERO);
```

Finally, we are ready to draw the reflected teapot:

```
Device->SetTransform(D3DTS_WORLD, &W);
Device->SetMaterial(&TeapotMtrl);
Device->SetTexture(0, 0);

Device->SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);
Teapot->DrawSubset(0);
```

The matrix `W` correctly transforms the reflected teapot into its appropriate position in the scene. See the previous section.
Also, the backface cull mode (see section 2.3.4: Backface Culling) is changed. We must do this because when an object is reflected, its front faces will be swapped with its back faces; however, the winding order will not be changed. Thus, the "new" front faces will have a winding order that indicates to Direct3D that they are back facing. Similarly, the "new" back-facing triangles will have a winding order that indicates to Direct3D that they are front facing. Therefore, to correct this, we must change our backface culling condition.

Cleanup

When done, we disable blending and stenciling and restore the usual cull mode:

```
Device->SetRenderState(D3DRS_ALPHABLENDENABLE, false);
Device->SetRenderState( D3DRS_STENCILENABLE, false);
Device->SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
} // end RenderMirror()
```

9.3 Sample: Planar Shadows

In this section we will see an implementation of planar shadows, a shadow which lies on a plane. These types of shadows are not as realistic as shadow volumes, but shadow volumes are an advanced topic. The DirectX SDK has an example demonstrating shadow volumes.
To implement planar shadows, we must first find the shadow that an object casts to a plane and model it geometrically so that we can render it. This can easily be done with some 3D math. We then render the polygons that describe the shadow with a black material at 50% transparency.
The example application for this section is called Stencil Shadow. It can be found in part 2, chapter 8.

9.3.1 Shadow Matrix

A shadow is essentially a parallel projection of an object onto the plane. We can represent the transformation from a vertex **p** to its projection **s** with the following shadow matrix:

```
P = normalize(Plane);
L = Light;
d = dot(P, L)

P.a * L.x + d  P.a * L.y      P.a * L.z      P.a * L.w
P.b * L.x      P.b * L.y + d  P.b * L.z      P.b * L.w
P.c * L.x      P.c * L.y      P.c * L.z + d  P.c * L.w
P.d * L.x      P.d * L.y      P.d * L.z      P.d * L.w + d
```

To see how to derive this matrix, see Chapter 6 of "Me and My (Fake) Shadow," from *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*.
The D3DX library provides the following function to build the shadow matrix given the plane that we wish to project the shadow to and a vector describing a parallel light if w = 0 or a point light if w = 1:

```
D3DXMATRIX *D3DXMatrixShadow(
      D3DXMATRIX *pOut,
      CONST D3DXVECTOR4 *pLight, // L
      CONST D3DXPLANE *pPlane // plane to cast shadow onto
);
```

9.3.2 Double Blending problem

To generate a shadow, we flatten out the geometry of an object onto the plane. When the shadow with transparency is rendered (using blending), the areas that have overlapping triangles will get blended multiple times and thus appear darker. See figure 9.2.

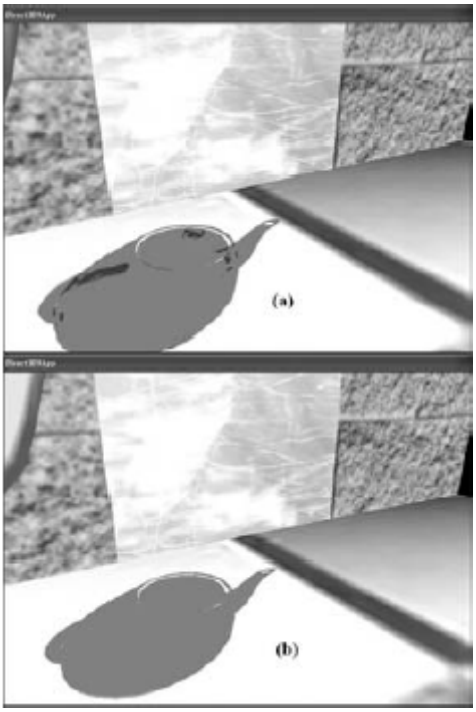


Figure 9.2: The teapot in (a) shows some black areas. These are areas where parts of the teapot overlap, causing a "double blend". The teapot in (b) is rendered without double blending.

Double blending can be solved with the stencil buffer. This can be done by marking the corresponding stencil buffer entries when the shadow's pixels are rendered to the back buffer. Then, if we attempt to write a pixel to an area that has already been rendered to (marked in the stencil buffer), the stencil test will fail. In this way, writing overlapping pixels is prevented and therefore avoids double blending artifacts.

9.3.3 Stencil Shadow example code

The code that generates the shadow is in the `RenderShadow` function.
We set the stencil render states first. The stencil comparison function is set to `D3DCMP_EQUAL` and the `D3DRS_STENCILREF` render state to 0x0, thereby specifying to render the shadow to the back buffer if the corresponding entry in the stencil buffer equals 0x0.
The stencil buffer is cleared to zero (0x0), which will make the stencil test always pass if we write a particular pixel of the shadow for the first time. Because we set `D3DRS_STENCILPASS` to `D3DSTENCILOP_INCR`, every time at a successful write of a pixel, its entry will be incremented to 0x1. The test will fail if we try to write to a pixel that we have already written to. This way, we avoided double blending.

```
void RenderShadow()
{
Device->SetRenderState(D3DRS_STENCILENABLE, true);
Device->SetRenderState(D3DRS_STENCILFUNC, D3DCMP_EQUAL);
Device->SetRenderState(D3DRS_STENCILREF, 0x0);
Device->SetRenderState(D3DRS_STENCILMASK, 0xffffffff);
Device->SetRenderState(D3DRS_STENCILWRITEMASK, 0xffffffff);
Device->SetRenderState(D3DRS_STENCILZFAIL, D3DSTENCILOP_KEEP);
Device->SetRenderState(D3DRS_STENCILFAIL, D3DSTENCILOP_KEEP);
Device->SetRenderState(D3DRS_STENCILPASS, D3DSTENCILOP_INCR);
```

Next, we compute the shadow transformation and translate the shadow into the appropriate place in the scene.

```
// compute the transformation to flatten the teapot into a shadow.
D3DXVECTOR4 lightDirection(0.707f, -0.707f, 0.707f, 0.0f);
D3DXPLANE groundPlane(0.0f, -1.0f, 0.0f, 0.0f);
```

```
D3DXMATRIX S;
D3DXMatrixShadow(&S, &lightDirection, &groundPlane);

D3DXMATRIX T;
D3DXMatrixTranslation(&T, TeapotPosition.x, TeapotPosition.y,
                      TeapotPosition.z);

D3DXMATRIX W = T * S;
Device->SetTransform(D3DTS_WORLD, &W);
```

Lastly, we set a black material at 50% transparency, disable depth testing, render the shadow, and then clean up by re-enabling the depth buffer and disabling alpha blending and stencil testing. We disable the depth buffer to prevent *z-fighting*, which is a visual artifact that occurs when two different surfaces have the same depth values in the depth buffer; the depth buffer doesn't know which should be in front of the other, and an annoying flicker occurs. Because the shadow and floor lie on the same plane, *z-fighting* between them will most likely occur. By rendering the floor first and the shadow after with depth testing disabled, we guarantee our shadow will be drawn over the floor.

```
Device->SetRenderState(D3DRS_ALPHABLENDENABLE, true);
Device->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
Device->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);

D3DMATERIAL9 mtrl = d3d::InitMtrl(d3d::BLACK, d3d::BLACK,
                                  d3d::BLACK, d3d::BLACK, 0.0f);
mtrl.Diffuse.a = 0.5f; // 50% transparency.

// Disable depth buffer so that z-fighting doesn't occur when we
// render the shadow on top of the floor.
Device->SetRenderState(D3DRS_ZENABLE, false);

Device->SetMaterial(&mtrl);
Device->SetTexture(0, 0);
Teapot->DrawSubset(0);

Device->SetRenderState(D3DRS_ZENABLE, true);

Device->SetRenderState(D3DRS_ALPHABLENDENABLE, false);
Device->SetRenderState(D3DRS_STENCILENABLE, false);

}//end RenderShadow()
```

9.4 Other usages for stenciling

Stenciling is not only used for mirrors, but can also be used for achieving other various effects. Examples can be outlines, silhouettes etc.
To get an outline of an object, you could apply a stencil mask to a primitive that's the same shape but slightly smaller than the primitive. The resulting image will contain only the primitive's outline. You can then fill this stencil-masked area of the primitive with a color or set of colors to produce an outline around the image.
For achieving a silhouette, set the stencil mask to the same size and shape as the primitive you're rendering, Direct3D will produce a final image containing a "black hole" where the primitive should be. By coloring this hole, you can produce a silhouette of the primitive.
See http://www.gamasutra.com/features/20000807/kovach_03.htm for more examples.

9.4 Summary

- The stencil buffer and depth buffer share the same surface and are therefore created at the same time. We specify the format of the depth/stencil surface using the `D3DFORMAT` types.
- Stenciling is used to block certain pixels from being rasterized. As we have seen in this chapter, this ability is useful for implementing mirrors and shadows among other applications.
- We can control stenciling operations and how the stencil buffer is updated through the `D3DRS_STENCIL*` render states.

10. A Flexible Camera Class

Thus far, we have used the `D3DXMatrixLookAtLH` function to compute a view space transformation matrix. This function is useful for a camera in a fixed position, but not so useful for a moving camera that reacts to user input. This chapter will show how to implement a `Camera` class suitable for e.g. first-person games.

Keywords: camera, up vector, right vector, position vector, look vector, pitch, yaw, roll, strafe, fly, move, transformation matrix, rotation, `D3DXMatrixRotationAxis`

10.1 Camera Design

We define a local coordinate system for the camera relative to the world coordinate system using four camera vectors: a right vector, up vector, look vector, and position vector, as figure 10.1 illustrates.

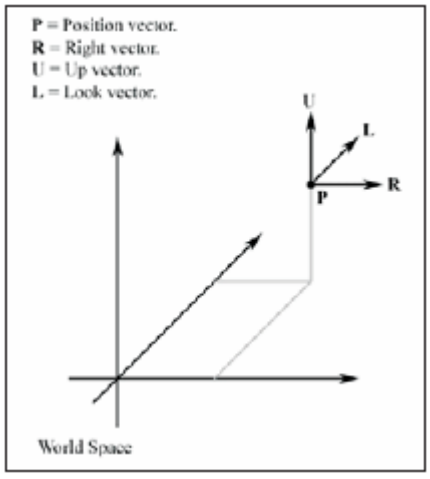


Figure 10.1: The camera vectors

The camera class will be able to perform the following six operations:

- Rotate around the right vector (pitch)
- Rotate around the up vector (yaw)
- Rotate around the look vector (roll)
- Strafe along the right vector
- Fly along the up vector
- Move along the look vector

The following shows a part of the `Camera` class where these six operations can be called.

```
class Camera {
     public:
          enum CameraType { LANDOBJECT, AIRCRAFT };
```

```
            Camera();
            Camera(CameraType cameraType);
            ~Camera();
            void strafe(float units); // left/right
            void fly(float units); // up/down
            void walk(float units); // forward/backward
            void pitch(float angle); // rotate on right vector
            void yaw(float angle); // rotate on up vector
            void roll(float angle); // rotate on look vector
            void getViewMatrix(D3DXMATRIX* V);
            void setCameraType(CameraType cameraType);
            void getPosition(D3DXVECTOR3* pos);
            void setPosition(D3DXVECTOR3* pos);
            void getRight(D3DXVECTOR3* right);
            void getUp(D3DXVECTOR3* up);
            void getLook(D3DXVECTOR3* look);
      private:
            CameraType _cameraType;
            D3DXVECTOR3  right;
            D3DXVECTOR3 _up;
            D3DXVECTOR3 _look;
            D3DXVECTOR3  pos;
};
```

The `Camera` class also provides a `CameraType` enumerated type. It has two modes: `LANDOBJECT` model and an `AIRCRAFT` model. The `AIRCRAFT` model allows us to move freely through space and gives us six degrees of freedom. However, in some games, such as a first-person shooter, we must restrict movement on certain axes. Specifying `LANDOBJECT` for the camera type will for example restrict a character from flying.

10.2 Implementation

10.2.1 The View Matrix

We now show how the view matrix transformation can be computed given the camera vectors. Let $\mathbf{p} = (p_x, p_y, p_z)$, $\mathbf{r} = (r_x, r_y, r_z)$, $\mathbf{u} = (u_x, u_y, u_z)$, and $\mathbf{d} = (d_x, d_y, d_z)$ be the position, right, up, and look vectors, respectively.

Recall that in Chapter 2 we said that the view space transformation transforms the geometry in the world so that the camera is centered at the origin and axis aligned with the major coordinate axes (see Figure 2.7).

We want a transformation matrix **V** such that:

pV = (0, 0, 0): The matrix V transforms the camera to the origin.
rV = (1, 0, 0): The matrix V aligns the right vector with the world x-axis.
uV = (0, 1, 0): The matrix V aligns the up vector with the world y-axis.
dV = (0, 0, 1): The matrix V aligns the look vector with the world z-axis.

The matrix **V** is defined as:

$$\mathbf{V} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -p_x & -p_y & -p_z & 1 \end{bmatrix} \begin{bmatrix} r_x & u_x & d_x & 0 \\ r_y & u_y & d_y & 0 \\ r_z & u_z & d_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} r_x & u_x & d_x & 0 \\ r_y & u_y & d_y & 0 \\ r_z & u_z & d_z & 0 \\ -\mathbf{p}\cdot\mathbf{r} & -\mathbf{p}\cdot\mathbf{u} & -\mathbf{p}\cdot\mathbf{d} & 1 \end{bmatrix}$$

The left matrix is the translation matrix and the right one the rotation matrix. These two matrices are combined and results in the matrix **V**. I will again omit the mathematics involved in how these matrices were generated. Call `Camera::getViewMatrix` to build the matrix **V**.

10.2.2 Rotation about an arbitrary axis

To rotate about an arbitrary axis, use the following D3DX function:

```
D3DXMATRIX *D3DXMatrixRotationAxis(
      D3DXMATRIX *pOut, // returns rotation matrix
      CONST D3DXVECTOR3 *pV, // axis to rotate around
      FLOAT Angle // angle, in radians, to rotate
);
```

Suppose we want to rotate π/2 radians around the axis defined by the vector (0.707, 0.707, 0). We would write:

```
D3DXMATRIX R;
D3DXVECTOR3 axis(0.707f, 0.707f, 0.0f);
D3DXMatrixRotationAxis(&R, &axis, D3DX_PI / 2.0f);
```

10.2.3 Pitch, Yaw and Roll

When we pitch, we need to rotate the up and look vectors around the right vector by the specified rotation angle. When we yaw, we need to rotate the look and right vectors around the up vector by the specified rotation angle. Finally, when we roll, we need to rotate the up and right vectors around the look vector by the specified rotation angle. The function `D3DXMatrixRotationAxis` is used for the rotation.
The pitch, yaw and roll functions are implemented as follows:

```
void Camera::pitch(float angle) {
      D3DXMATRIX T;
      D3DXMatrixRotationAxis(&T, &_right, angle);

      // rotate _up and _look around _right vector
      D3DXVec3TransformCoord(&_up,&_up, &T);
      D3DXVec3TransformCoord(&_look,&_look, &T);
}

void Camera::yaw(float angle) {
      D3DXMATRIX T;

      // rotate around world y (0, 1, 0) always for land object
      if( _cameraType == LANDOBJECT )
            D3DXMatrixRotationY(&T, angle);

      // rotate around own up vector for aircraft
      if( _cameraType == AIRCRAFT )
            D3DXMatrixRotationAxis(&T, &_up, angle);

      // rotate _right and _look around _up or y-axis
      D3DXVec3TransformCoord(&_right,&_right, &T);
      D3DXVec3TransformCoord(&_look,&_look, &T);
}

void Camera::roll(float angle) {
      // only roll for aircraft type
      if( _cameraType == AIRCRAFT ) {
            D3DXMATRIX T;
            D3DXMatrixRotationAxis(&T, &_look, angle);
```

```
            // rotate  up and  right around  look vector
            D3DXVec3TransformCoord(&_right,&_right, &T);
            D3DXVec3TransformCoord(& up,& up, &T);
      }
}
```

The roll operation is only done for the `AIRCRAFT` camera type. This is because it doesn't feel right if a land object yaws when tilted. You could of course change this behavior in the `Camera` class.

10.2.4 Walking, Strafing and Flying

Walking means moving in the direction that we are looking (that is, along the look vector). Strafing is moving side to side from the direction we are looking, which is of course moving along the right vector. Finally, we say that flying is moving along the up vector. To move along any of these axes, we simply add a vector that points in the same direction as the axis that we want to move along to our position vector.
The `walk`, `strafe`, and `fly` methods implementation:

```
void Camera::walk(float units) {
      // move only on xz plane for land object
      if( _cameraType == LANDOBJECT )
            _pos += D3DXVECTOR3( _look.x, 0.0f, _look.z) * units;
      if(  cameraType == AIRCRAFT )
            _pos += _look * units;
}

void Camera::strafe(float units) {
      // move only on xz plane for land object
      if(  cameraType == LANDOBJECT )
            _pos += D3DXVECTOR3( right.x, 0.0f, _right.z) * units;
      if(  cameraType == AIRCRAFT )
            pos +=  right * units;
}

void Camera::fly(float units) {
      if( _cameraType == AIRCRAFT )
            pos +=  up * units;
}
```

10.3 Sample: Camera

The Camera application can be found at part 3, chapter 12. This sample uses the `Camera` class which was discussed in this chapter. The following keys are implemented:

- W/S: Walk forward/backward
- A/D: Strafe left/right
- R/F: Fly up/down
- Up/Down arrow keys: Pitch
- Left/Right arrow keys: Yaw
- N/M: Roll

These keys are handled in the `Display` function.
The global `Camera` object `TheCamera` is instantiated at the global scope. Also notice that we move the camera with respect to the time change (`timeDelta`); this keeps us moving at a steady speed independent of the frame rate.

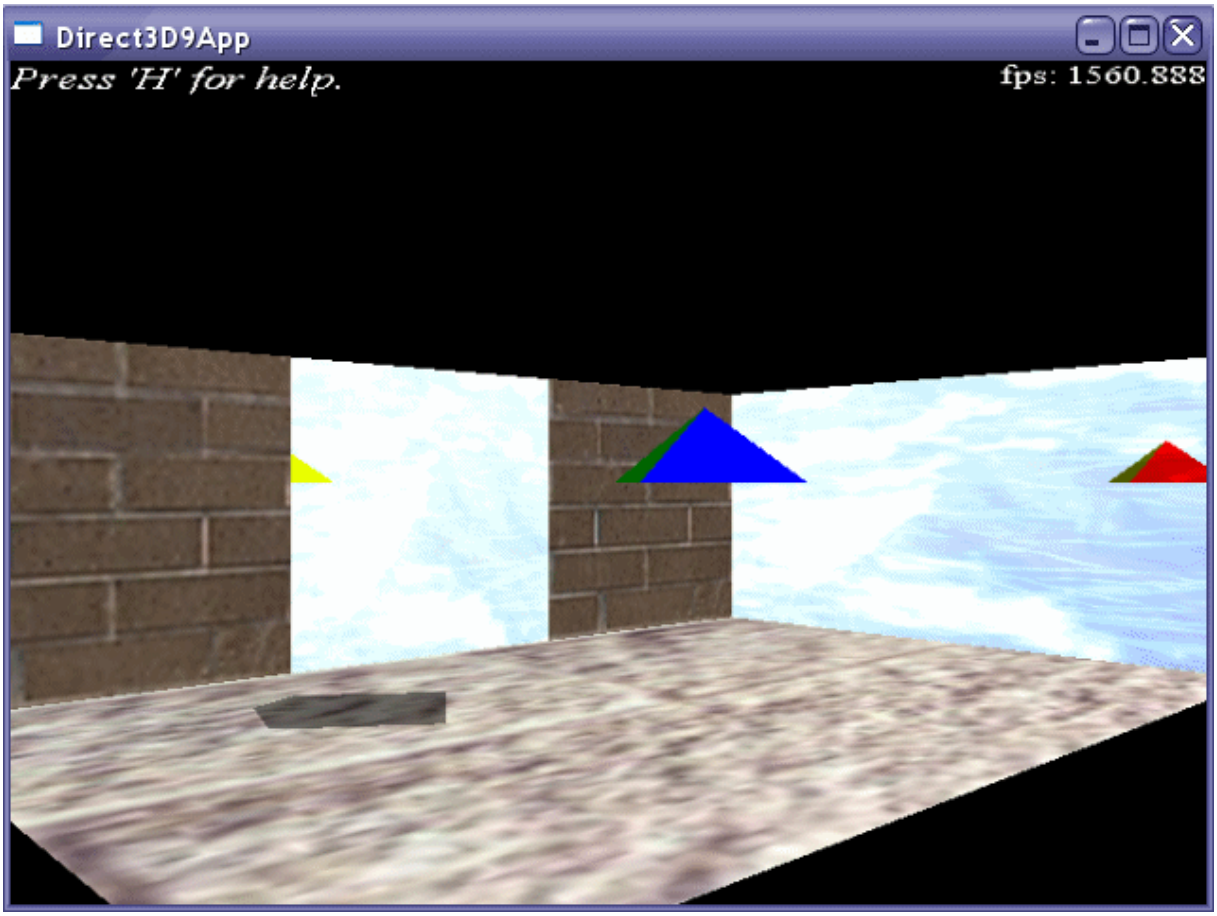
```
bool Display(float timeDelta) {
      if( Device ) {
            //
            // Update: Update the camera.
            //
            if( ::GetAsyncKeyState('W') & 0x8000f )
                  TheCamera.walk(4.0f * timeDelta);
            if( ::GetAsyncKeyState('S') & 0x8000f )
                  TheCamera.walk(-4.0f * timeDelta);
            if( ::GetAsyncKeyState('A') & 0x8000f )
                  TheCamera.strafe(-4.0f * timeDelta);
            if( ::GetAsyncKeyState('D') & 0x8000f )
                  TheCamera.strafe(4.0f * timeDelta);
            if( ::GetAsyncKeyState('R') & 0x8000f )
                  TheCamera.fly(4.0f * timeDelta);
            if( ::GetAsyncKeyState('F') & 0x8000f )
                  TheCamera.fly(-4.0f * timeDelta);
            if( ::GetAsyncKeyState(VK_UP) & 0x8000f )
                  TheCamera.pitch(1.0f * timeDelta);
            if( ::GetAsyncKeyState(VK_DOWN) & 0x8000f )
                  TheCamera.pitch(-1.0f * timeDelta);
            if( ::GetAsyncKeyState(VK_LEFT) & 0x8000f )
                  TheCamera.yaw(-1.0f * timeDelta);
            if( ::GetAsyncKeyState(VK_RIGHT) & 0x8000f )
                  TheCamera.yaw(1.0f * timeDelta);
            if( ::GetAsyncKeyState('N') & 0x8000f )
                  TheCamera.roll(1.0f * timeDelta);
            if( ::GetAsyncKeyState('M') & 0x8000f )
                  TheCamera.roll(-1.0f * timeDelta);

            // Update the view matrix representing the cameras
            // new position/orientation.
            D3DXMATRIX V;
            TheCamera.getViewMatrix(&V);
            Device->SetTransform(D3DTS_VIEW, &V);

            //
            // Render
            //
            Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                              0x00000000, 1.0f, 0);

            Device->BeginScene();
            d3d::DrawBasicScene(Device, 1.0f);
            Device->EndScene();

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

10.4 Summary

- We describe the position and orientation of our camera in the world coordinate system by maintaining four vectors: a right vector, an up vector, a look vector, and a position vector. With this description, we can easily implement a camera with six degrees of freedom, giving a flexible camera interface

11. Sample Project

The final sample is a self provided demo application. I have tried to implement as many techniques, which were discussed in this document, as possible. All techniques except for colors were actually used.
The sample is not really a fun game, but shows a nice screen with a rotating model, pyramid or teapot, two mirrors and a shadow. The controls can be inspected by pressing the 'H' button. See the screenshot.
The sample can be found at: http://www.cs.vu.nl/~tljchung/directx/samples/Project.zip.



11.1 Implementation

The following was used for this sample:

- The pyramid: Used vertex buffer and index buffer, drawn with `DrawIndexedPrimitive`. See chapter 3 Drawing.
- Teapot: Generated with the D3DX library. `D3DXCreateTeapot` and drawn with `DrawSubset`. See chapter 3.
  Press 'M' to switch between the pyramid and the teapot.
- Light: Enabled a white directional light. The models use some of the materials provided in the d3dUtility.h file. The Vertex format has normals added for this purpose. See chapter 5.
- Textures: Applied textures on the floor, mirrors, walls and the bottom of the pyramid. The pyramid bottom has a crate texture. The Vertex format has been added texture coordinates _u and _v for textures. See chapter 6.
- Blending: Was needed for the mirrors and the shadow. See chapter 7.
- Fonts: Used ID3DXFont to draw the help messages and the fps. See chapter 8.

- Stenciling: For the mirrors and shadow. See chapter 9.
- Camera: Used the `Camera` class from chapter 10.
- Rotation: Added rotations using a global rotation matrix `ROT`. This matrix is multiplied with the matrix provided to `SetTransform`.

```
Device->SetTransform(D3DTS_WORLD, &(ROT * W));
```

Bibliography

- Luna, Frank. Introduction to 3D GAME Programming with DirectX 9.0. Wordware Publishing, Inc, 2003.
  http://www.moon-labs.com/
- Kovach, Peter. Gamasutra: Inside Direct3D: Stencil Buffers [08.07.00].
  http://www.gamasutra.com/features/20000807/kovach_01.htm
- Microsoft. DirectX 9.0 SDK documentation.
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/directx9cpp.asp