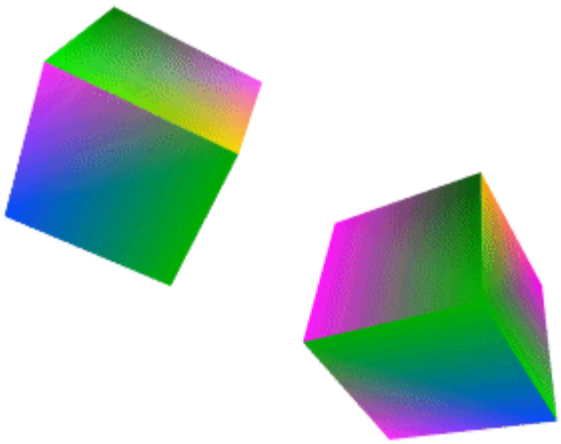
Workbook Chapter One:
3D Graphics Fundamentals



Getting Started with DirectX Graphics

Veteran programmers, and even veteran games players, will surely remember the days when all games ran on top of a low-level text based operating system known as DOS™. At the time, Microsoft Windows™ had become a respectable platform to run business applications but was generally considered a very poor choice for developing cutting edge 3D games. The problem was that the platform isolated the programmer from the underlying graphics hardware by using a software layer called the Graphics Device Interface (GDI). This interface contained a robust collection of 2D text and primitive drawing functions that the developer could use to render 2D output to the screen. In some ways this was advantageous because it meant developers did not have to concern themselves with issues such as which chipset was used by the graphics card in the end user's system. To the developer it all *looked* the same. If you wanted to draw a rectangle, you would simply instruct GDI to draw that rectangle. GDI would handle the interaction with the graphics hardware to produce the physical output.

The GDI was built to be stable and robust. Unfortunately this came at the cost of prohibiting the developer direct access to the screen and video memory. This situation is generally unacceptable for game development projects because drawing operations had to be converted by GDI into native instructions that the graphics hardware could understand. This heavy software abstraction layer between the developer and the hardware rendering was very slow; so slow in fact that it could not seriously be used for modern games.

Games running through DOS had no such limitation. The graphics hardware could be controlled directly by the programmer using low-level techniques and games could run much faster. Despite these benefits, DOS games were a challenge. This was true not only for the developer but also for the game player.

From the developer's perspective, the PC had become so popular that many manufacturers produced graphics cards, all with different chipsets, each of which often spoke different languages. This meant developers had to make sure their games worked on many different types of hardware. There was no standard rendering API at that time. Because each graphics card had to be uniquely programmed, developers often had to create many different versions of drawing functions to work with the different graphics hardware. If new hardware was released after the software application was released there was a good chance the application would not work with that hardware. This also presented a difficulty for the bedroom programmer (the hobbyist) because they generally did not have the budget to purchase all of the available graphics hardware on the market to ensure that their game worked on all of them.

From the perspective of the games player, many felt it too difficult just to get a game to install correctly. The user would often be quizzed about the chipset they were using on their graphics card and the amount of available video memory they had. This may not sound like such a big deal to a technical person, but many people who were not computer savvy did not really understand what all these terms meant or even exactly what hardware they had inside their system. Software companies had to provide extensive customer support as an added expense. This was in contrast to games consoles such as the Super Nintendo™, where even a young child could play a game simply by inserting a cartridge.

Microsoft realized that this problem had to be addressed if they wanted Windows to become a dominant gaming platform. So shortly after the release of Windows 95, Microsoft released a royalty-free multimedia development library called 'The Game SDK'. This was essentially version 1.0 of DirectX. The name DirectX however, was officially adopted along with version 2.0 of this SDK, likely because it had matured into a full-blown multimedia library and was no longer limited to just games development. Although the earlier versions of DirectX were somewhat rough around the edges, it has matured greatly over the years. From DirectX 5 onwards, developers really started to sit up and take notice. Now we are at version 9 of DirectX and it really is amazing how far it has come in such a short period of time.

DirectX provided the answers to many of the problems that had plagued the development of games and entertainment titles up until that point. First, it was designed for the Windows platform. This meant that developers could create their games in an environment where Win32 API features (such as multi-threading and user interfaces) were already available. Second, it provided a unified API, much like the GDI had done before, but this time it was very fast. The developer no longer had to worry about what graphics hardware the end user would be playing on, and could usually leave it up to DirectX to communicate with the hardware correctly. This was accomplished through the use of **driver programs**. Graphics cards that support DirectX (which is virtually all of them now) come with a driver which is installed on the end user's system. This driver is written by the card manufacturer and is a very thin and fast software layer that takes the requests passed through various DirectX functions by the application, and turns them into instructions that the hardware understands. This means DirectX can talk to all graphics cards as though they are the same even when they are radically different from one another. Drivers supplied by the card manufacturer handle the conversion into hardware specific instructions very quickly. One of the other advantages that DirectX affords us (over the GDI) is that it does not completely isolate us from the end user's hardware.

DirectX also takes advantage of 3D hardware acceleration without requiring any additional code from the developer. If you render a triangle using DirectX, and the computer running the application has a 3D accelerated graphics card, DirectX will use those features to render that triangle at high speeds. The latest 3D hardware also accelerates 3D mathematics (which was always the domain of the CPU in the past). This means that many graphics card can handle the thousands of mathematical calculations needed to render a scene whilst leaving the CPU free to handle other tasks such as artificial intelligence or other game specific tasks.

The DirectX API

DirectX is divided into several code modules or Application Programming Interfaces (APIs). Each covers different areas of multimedia development. Some of the DirectX APIs are listed below along with a brief description of the functionality they provide to the developer. Although this course is primarily focused on DirectX Graphics, it is useful to have a broader picture of the entire DirectX multimedia library:

DirectX Graphics

In older versions of DirectX, 2D and 3D operations were divided among two APIs called DirectDraw and Direct3D respectively. From DirectX 8.0 onwards, these APIs were merged into a single API called DirectX Graphics. Many people still refer to DirectX Graphics as Direct3D. As you will see, most of DirectX Graphics functions and interfaces usually start with D3D (short for Direct3D) so in many ways this makes some sense. The terms 'Direct3D' and 'DirectX Graphics' will be used interchangeably from this point on. If we mention either of these terms it is to be assumed that we are talking about the same API: DirectX Graphics.

DirectX Audio

The DirectX Audio API contains functionality for managing and playing audio samples and music within your application. It includes support for three dimensional / positional audio, and also includes support for hardware sound processing and environmental effects. DirectX Audio was previously split into two APIs, known as DirectSound and DirectMusic but following the release of DirectX 8.0 they have been merged into one. This API is **not** covered in this course.

DirectInput

The DirectInput API contains functionality to handle user-input peripherals. It provides functions for managing and reading devices such as Joysticks, Game Pads, and Force Feedback Wheels as well as the keyboard and the mouse. The Game Institute offers a course covering the full DirectInput API so be sure to check out the course offerings page at www.gameinstitute.com for more information as you continue to build out your own projects.

DirectPlay

This API provides functionality generally used in the implementation of networked multiplayer games and similar applications. It includes support for transmitting and receiving data across many different types of network environments, including the Internet. As with most aspects of DirectX, this API is designed as an application layer which unifies the system used to transmit and receive data regardless of the underlying network infrastructure. The Game Institute also provides training in this API so be sure to check out this course when you decide to add network capability to your game projects.

DirectShow

The DirectShow API provides features which encapsulate the recording and playback of high quality multi-media streams. This includes support for many popular formats such as MPEG, AVI, ASF and MP3 audio files. This API is **not** covered in this course.

Direct Setup

This API provides you with a straightforward way to distribute and install the DirectX runtime libraries on the end user's machine. You may have seen this in action many times before when you installed a new game that uses a more recent version of DirectX than the one you currently have installed. When this is found to be the case you are often informed that you need the later version of DirectX, after which the actual installation proceeds. This requires much more than a few file copy operations, so you should make sure that you use this API to install DirectX on the end user machine when your game is finally shipped. This API is **not** covered in this course.

Installing the DirectX9 SDK

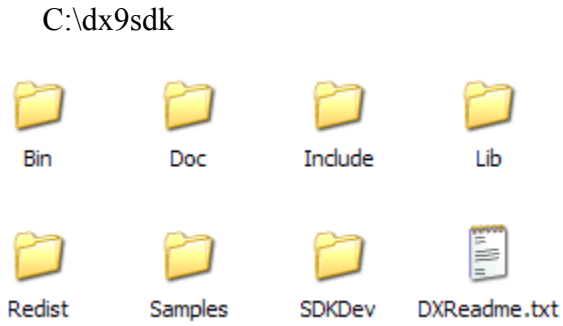
In order to use DirectX Graphics and the D3DX utility extension, we need to set up our compiler so that it can find the DX9 header files and the DX9 library files. We will need to include the d3d9.lib and d3dx9.lib library files within all of the projects that make use of DirectX Graphics. We must also include the d3dx9.h header file at the top of the source files that require their functionality (a common header file could also be used). When using d3dx9.h we do not have to manually include in d3d9.h as this is included automatically when including d3dx9.h.

Let us first cover setting up the DX9 SDK for your compiler. The following examples are for Microsoft's Visual C++ 6 compiler. If you are using a different compiler then you will have to interpret and translate the following instructions for use with your particular system.

The first thing you will need to do is to visit the Microsoft website (www.microsoft.com) and download the DirectX9 software development kit (SDK). This is a fairly sizable download especially for people using 56k dial up accounts (around 200MB). If you are unable to download files this big, Microsoft provides a means to purchase the DirectX 9 SDK on CD from their website (for a minimal charge that basically covers postage, packaging, and shipping).

Once the file has been downloaded (or you have received the package on CD), run the setup executable. This will install the SDK on your computer. In the following example, we have installed the SDK in the folder "C:\DX9SDK". If you decide to place it elsewhere on your system, you must change the path used in the following examples to match the folder into which you decided to install it.

Once the SDK has been installed (and you have rebooted your machine) you will find that a folder has been created ('C:\DX9SDK' in this example) with several sub-folders. The sub-folders of importance are shown below:



Bin
The 'Bin' folder contains utility applications that aid in the development of DX9 applications. These are incredibly useful tools to have at your disposal. Some worthy of mention are:

a. ***DXCapsViewer.exe***: Allows you to see all of the DirectX features and modes supported by your current hardware. Video modes, refresh rates, and texture blending operations are some examples.

b. ***DXErr.exe:*** Allows you to enter error codes returned by DX API functions and retrieve a meaningful description to help you to diagnose what went wrong.

c. ***DXTex.exe:*** Allows you to import bitmaps that are to be used as textures, and convert them to the DirectX native texture format known as .DDS. You do not have to use DDS files but they can be convenient in certain circumstances.

d. ***vsa.exe:*** Allows you to compile vertex shaders.

e. ***psa.exe:*** Allows you to compile pixel shaders.

Doc
This folder contains your lifeline to DirectX Graphics development (ok, perhaps your *second* lifeline, after this course). It contains the complete reference manual for DirectX packed with hundreds of pages of information. You will no doubt use this as a reference time and time again. Every possible function call, interface, structure, and macro used by DirectX is explained to some degree in here.

Include
This folder contains the entire set of C++ header files that you will need to include in your project to create a DirectX application. We will discuss shortly how to set up the search paths used by the development environment so that the compiler automatically uses this folder when building your project.

Libs
This folder contains all of the library files that you will need to link into your project in order to gain access to DirectX functions and interfaces. We will show you how to set the environment up in a moment and discuss which lib files you need to link into your project and when.

Redist
This folder contains the distributable DirectX runtime which you can ship to the end user along with your application. The executable in this folder allows for the automated version checking and installation of the DirectX9 runtime on the end user system. There are examples of how to use this system correctly in the samples folder.

Samples
The samples folder is another invaluable resource when it comes to learning DirectX programming. It contains dozens of example programs (with source code) showing how to use DirectX and all of its features. This folder also contains precompiled binaries so that you can run the samples without being

required to build the source. This is a good way to test that DirectX9 is correctly installed to your computer.

SDKDev

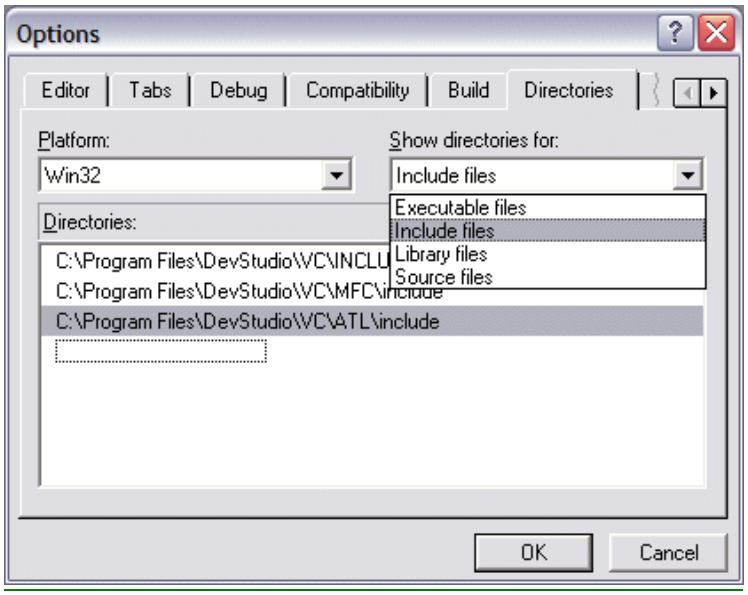
This directory contains the runtime install applications that are automatically installed with the SDK. They are English language only and contain both debug and retail DirectX 9.0 system components. You can switch between the retail and debug versions of the runtime via the DirectX Control Panel component (accessible via the Windows Control Panel). You can use the debug runtime to receive additional debug information from DirectX via the C++ IDE. If the control panel icon is not available, try re-installing the debug runtime contained in this folder. These installers are not for redistribution, and are designed for SDK development only.

Note: If you choose to install the debug runtimes, please make sure that you disable it via the control panel whenever you do not require additional debug information. The debug runtimes are significantly slower than the retail runtimes.

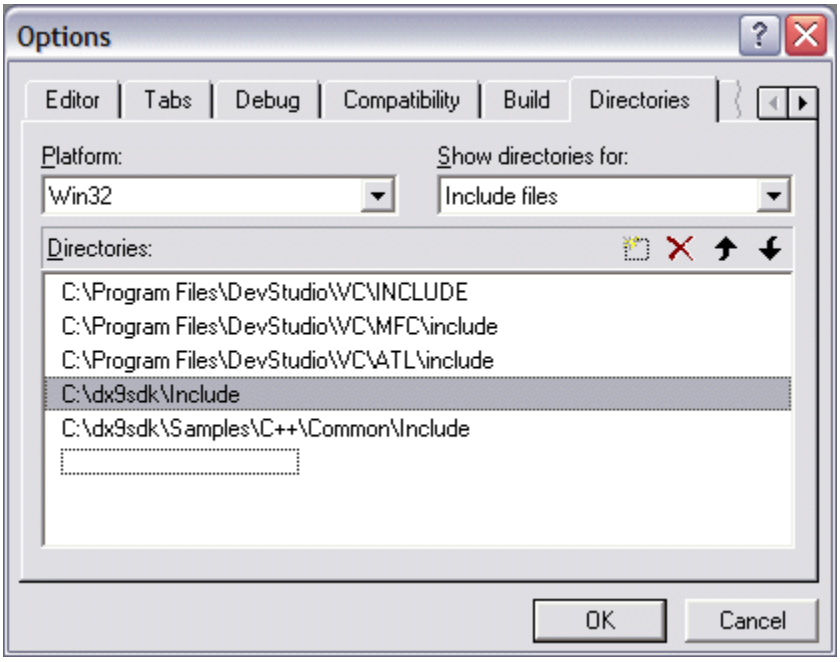
Setting up the Build Environment for DirectX9

Setting up the environment is easy if you are using Microsoft Visual C++ 6. If you are not using Microsoft VC++ 6 then you will need to translate the following instructions to work with your preferred compiler/environment.

The first thing we will do is setup the IDE so that it will search the "C:\DX9SDK\Include" folder automatically when searching for header files. This is done via the ***Tools*** / ***Options*** menu item which will bring up the options property sheet. Next you need to click on the ***Directories*** **tab** as demonstrated below:



Select ‘Include files’ from the drop down combo box in order to display a list of all the folders currently in the environment include search path. Whenever a ‘`#include <file>`’ directive is encountered within your code, the compiler will search for the file in each of the folders listed (in order) until it is found. We need to add the folder in which our DirectX9 headers files are contained:



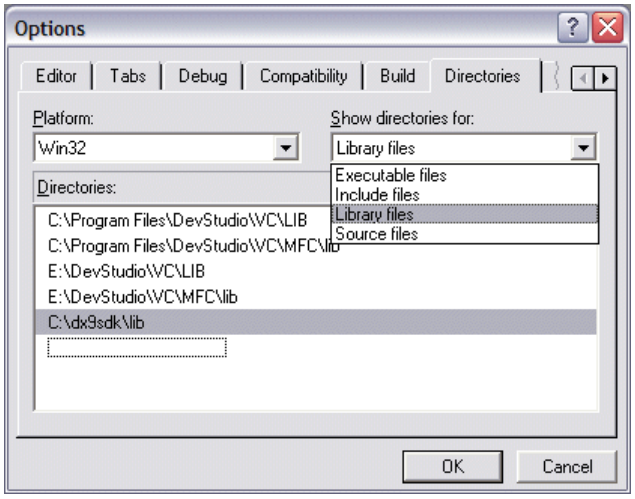
In the above image we have added two folders to the list. The first one (described earlier) is the folder in which the primary include files are contained. These are required for building a DirectX application.

The second path we have added contains many of the include files used by certain SDK sample applications. These include the header files for the SDK Framework. The framework is a series of classes that can be used to define a pre-built code structure for your DirectX programs. It provides certain benefits such as functionality for handling the environment setup, texture import and manipulation, and so on. We will not be using the framework in this course, although you can make use of individual components if you desire.

Warning:
The search path list is processed in order from top to bottom. This is important to note if you have a previous version of the SDK installed and you have a path to those folders in the list. If it is higher up in the list and some of those files share the same name as the ones in the dx9sdk, those files will be processed first and used to build your executable. This is not a good thing. To change the order of the directories, simply select the path item you want to move and drag it up or down in the list.

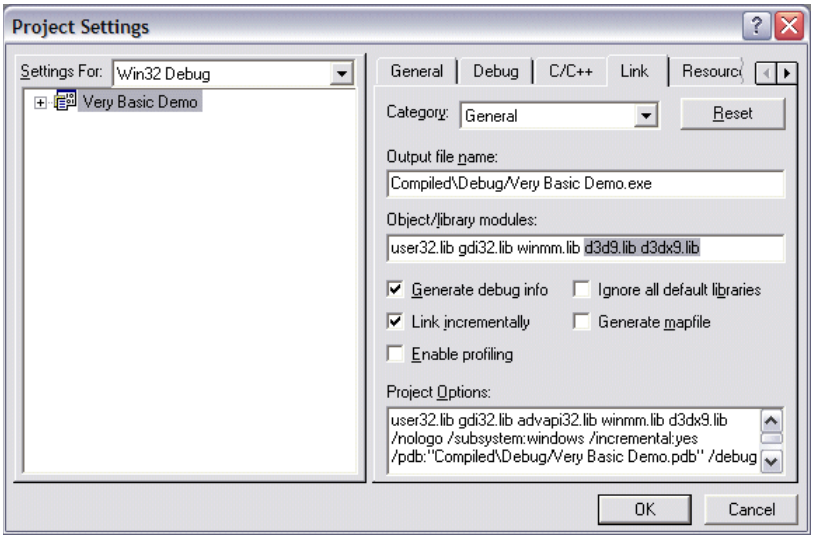
Our next task is to add another search path in the same way described above. This time we want to add it to the directories checked whilst searching for library files. Simply pull down the combo box as

before and select ***library files***. The current list of search directories will be displayed. As before, we need to add a search path so that the compiler searches the 'C:\DX9SDK\Lib' folder shown below:



Again, the search order is significant. If you have legacy lib files (from an older SDK installation) in different search paths that share the same name as some of the DirectX 9 lib files, then you will experience problems during compilation. So, make sure that the priority listings are at the top.

The last step is informing the environment about which DirectX library files we would like linked with our application. To use DirectX Graphics we need to link in two library files, '***d3d9.lib***' which contains the core DirectX Graphics functionality and '***d3dx9.lib***' which contains the D3DX helper library. We will set this list up on a per project basis. For the project files accompanying this lesson, this will have been done already. If you are starting a new project, you will need to carry out this procedure via the '***Projects*** / ***Settings***' menu item. This will open the project settings property sheet. When this happens select the '***Link***' tab to display the settings for the linker as shown:



As you can see in the above image we have added the names of the two DirectX library files we need to the end of the 'Object/Library Modules' list.

Lab Project 1.1: The Transformation Pipeline

Our first demonstration application will be a simple wire frame software transformation and rendering application that animates two spinning cubes. We will use the Window GDI to draw the lines for each polygon.

We have chosen to use a class to store vertex information in this example but you could also use a struct. The class contains three floating point member variables that describe the offset from the origin along each respective axis. It also has a constructor which receives X, Y, and Z values to aid in the easy initialization of the vertex. Although it is considered more OOP correct to make the data members private and to provide accessor functions that read and set the variables, in the interest keeping things simple and to minimize code, we will make the members public. This is actually typical for a vertex class since the values may need to be accessed many times in very tight code loops. The overhead of calling functions such as 'SetVertexX(value)' and 'GetVertexX()' might be significant where in-lining cannot be used.

```
class CVertex
{
public:
   // Constructors
   CVertex( float fX, float fY, float fZ);
   CVertex();

   // Public Variables for This Class
   float       x;          // Vertex X Coordinate
   float       y;          // Vertex Y Coordinate
   float       z;          // Vertex Z Coordinate
};
```

The next class we need will store a polygon. Since each polygon is made up of a number of vertices, our polygon structure will look like this:

```
class CPolygon
{
public:
    // Constructors & Destructors for This Class.
              CPolygon( USHORT VertexCount );
              CPolygon();
    virtual ~CPolygon();

    // Public member functions
    long        AddVertex( USHORT Count = 1 );

    // Public member variables
    USHORT      m_nVertexCount;  // Number of vertices stored.
    CVertex    *m_pVertex;       // Simple vertex array
};
```

The Polygon class has a member variable ***m_nVertexCount*** which will store the number of vertices used to define this polygon. In our example all polygons are cube faces that have four corner points

and therefore the vertex count for each of our polygons will be 4. The ***CVertex*** pointer will be used to allocate an array for the number of vertices required for this polygon.

The default constructor simply initializes the member variables to zero or null:

```
// Default Constructor
CPolygon::CPolygon()
{
     m_nVertexCount  = 0;
     m_pVertex       = NULL;
}
```

The second constructor allows us to pass in the number of vertices to be allocated. This function calls the member function ***AddVertex*** to allocate the actual vertex memory.

```
// Constructor 2
CPolygon::CPolygon( USHORT Count )
{
    // Reset / Clear all required values
    m_nVertexCount  = 0;
    m_pVertex       = NULL;

    // Add vertices
    AddVertex( Count );
}
```

The destructor simply deletes the vertex array if one exists.

```
// Destructor
CPolygon::~CPolygon()
{
    // Release our vertices
    if ( m_pVertex ) delete []m_pVertex;

    // Clear variables
    m_pVertex       = NULL;
    m_nVertexCount  = 0;
}
```

The AddVertex function allocates a new block of memory large enough to hold both the requested number of vertices and those already existing inside the polygon. Data is copied from the old vertex array into the new one and the old array discarded. The additional vertices that have been added to the end of the array will be initialized to the values specified in the default CVertex constructor.

```
long CPolygon::AddVertex( USHORT Count )
{
    CVertex * pVertexBuffer = NULL;

    // Allocate new resized array
    if (!( pVertexBuffer = new CVertex[ m_nVertexCount + Count ] )) return -1;

    // Existing Data?
```

```
    if ( m_pVertex )
    {
        // Copy old data into new buffer
        memcpy( pVertexBuffer, m_pVertex, m_nVertexCount * sizeof(CVertex) );

        // Release old buffer
        delete []m_pVertex;

    } // End if

    // Store pointer for new buffer
    m_pVertex = pVertexBuffer;
    m_nVertexCount += Count;

    // Return first vertex
    return m_nVertexCount - Count;
}
```

CMesh

The CMesh class will manage a collection of polygons. In our class we have chosen to store an array of polygon pointers. We also need a member variable that tells us how many polygons the mesh contains. Our cube mesh will use eight polygons. Of course, we will not hard-code such limitations so that we can reuse these classes later to store polygons with more than 4 vertices (hexagons for example) or meshes with thousands of polygons.

```
class CMesh
{
public:
    // Constructors & Destructors
             CMesh( ULONG Count );
             CMesh();
    virtual ~CMesh();

    // Public Functions
    long        AddPolygon( ULONG Count = 1 );

    // Public Member Variables
    ULONG       m_nPolygonCount;        // Number of polygons stored
    CPolygon  **m_pPolygon;             // Simply polygon array

};



// Default constructor

CMesh::CMesh()
{

    m_nPolygonCount = 0;
    m_pPolygon      = NULL;
}
```

The second constructor allows us to specify how many polygons we want automatically allocated. It wraps the ***AddPolygon*** function which is where the allocation takes place.

```
CMesh::CMesh( ULONG Count )
{
    m_nPolygonCount = 0;
    m_pPolygon      = NULL;
    // Add Polygons
    AddPolygon( Count );
}
```

The destructor releases any memory that has been allocated. This involves releasing all polygons owned by the mesh.

```
CMesh::~CMesh()
{
    // Release our mesh components
    if ( m_pPolygon )
    {
        // Delete all individual polygons in the array.
        for ( ULONG i = 0; i < m_nPolygonCount; i++ )
        {
            if ( m_pPolygon[i] ) delete m_pPolygon[i];

        } // Next Polygon

        // Free up the array itself
        delete []m_pPolygon;

    } // End if

    // Clear variables
    m_pPolygon      = NULL;
    m_nPolygonCount = 0;
}
```

Next we look at the polygon allocation function ***AddPolygon***. The CMesh contains an array of polygon pointers. This makes resizing the arrays easier when a new polygon is added.

```
long CMesh::AddPolygon( ULONG Count )
{
    CPolygon ** pPolyBuffer = NULL;

    // Allocate new resized array
    if (!( pPolyBuffer = new CPolygon*[ m_nPolygonCount + Count ] )) return -1;

    // Clear out slack pointers
    ZeroMemory( &pPolyBuffer[ m_nPolygonCount ], Count * sizeof( CPolygon* ) );

    // Existing Data?
    if ( m_pPolygon )
    {
        // Copy old data into new buffer
        memcpy( pPolyBuffer, m_pPolygon, m_nPolygonCount * sizeof( CPolygon* ) );
        // Release old buffer
```

```
        delete []m_pPolygon;

    } // End if

    // Store pointer for new buffer
    m_pPolygon = pPolyBuffer;

    // Allocate new polygon pointers
    for ( UINT i = 0; i < Count; i++ )
    {
        // Allocate new poly
        if (!( m_pPolygon[ m_nPolygonCount ] = new CPolygon() )) return -1;

        // Increase overall poly count
        m_nPolygonCount++;

    } // Next Polygon

    // Return first polygon
    return m_nPolygonCount - Count;
}
```

With these classes in place we add a function call at the start of our application to initialize the mesh object and fill it with the vertices of our cube. A function that used our new classes would look something like the following (assuming that g_Mesh is a global CMesh object variable):

```
bool BuildObjects()
{
    CPolygon * pPoly = NULL;

    // Add 6 polygons to this mesh.
    if ( g_Mesh.AddPolygon( 6 ) < 0 ) return false;

    // Front Face
    pPoly = g_Mesh.m_pPolygon[0];
    if ( pPoly->AddVertex( 4 ) < 0 ) return false;

    pPoly->m_pVertex[0] = CVertex( -2,  2, -2 );  // P1
    pPoly->m_pVertex[1] = CVertex(  2,  2, -2 );  // P4
    pPoly->m_pVertex[2] = CVertex(  2, -2, -2 );  // P8
    pPoly->m_pVertex[3] = CVertex( -2, -2, -2 );  // P5

     // Top Face
    pPoly = g_Mesh.m_pPolygon[1];
    if ( pPoly->AddVertex( 4 ) < 0 ) return false;

    pPoly->m_pVertex[0] = CVertex( -2,  2,  2 );  // P2
    pPoly->m_pVertex[1] = CVertex(  2,  2,  2 );  // P3
    pPoly->m_pVertex[2] = CVertex(  2,  2, -2 );  // P4
    pPoly->m_pVertex[3] = CVertex( -2,  2, -2 );  // P1

    // Back Face
    pPoly = g_Mesh.m_pPolygon[2];
    if ( pPoly->AddVertex( 4 ) < 0 ) return false;

    pPoly->m_pVertex[0] = CVertex( -2, -2,  2 );  // P6
```

```
    pPoly->m_pVertex[1] = CVertex(  2, -2,  2 );  // P7
    pPoly->m_pVertex[2] = CVertex(  2,  2,  2 );  // P3
    pPoly->m_pVertex[3] = CVertex( -2,  2,  2 ),  // P2

    // Bottom Face
    pPoly = g_Mesh.m_pPolygon[3];
    if ( pPoly->AddVertex( 4 ) < 0 ) return false;

    pPoly->m_pVertex[0] = CVertex( -2, -2, -2 );  // P5
    pPoly->m_pVertex[1] = CVertex(  2, -2, -2 );  // P8
    pPoly->m_pVertex[2] = CVertex(  2, -2,  2 );  // P7
    pPoly->m_pVertex[3] = CVertex( -2, -2,  2 );  // P6

    // Left Face
    pPoly = g_Mesh.m_pPolygon[4];
    if ( pPoly->AddVertex( 4 ) < 0 ) return false;

    pPoly->m_pVertex[0] = CVertex( -2,  2,  2 );  // P2
    pPoly->m_pVertex[1] = CVertex( -2,  2, -2 );  // P1
    pPoly->m_pVertex[2] = CVertex( -2, -2, -2 );  // P5
    pPoly->m_pVertex[3] = CVertex( -2, -2,  2 );  // P6

    // Right Face
    pPoly = g_Mesh.m_pPolygon[5];
    if ( pPoly->AddVertex( 4 ) < 0 ) return false;

    pPoly->m_pVertex[0] = CVertex(  2,  2, -2 );  // P4
    pPoly->m_pVertex[1] = CVertex(  2,  2,  2 );  // P3
    pPoly->m_pVertex[2] = CVertex(  2, -2,  2 );  // P7
    pPoly->m_pVertex[3] = CVertex(  2, -2, -2 );  // P8

    // Success!
    return true;
}
```

WinMain

A WinMain function will typically call initialization routines and then enter a message loop that is continuously processed until the application exits. Our WinMain function will be very simple because we have moved the message pump handler into a class called CGameApp. That class will be responsible for managing the entire application.

```
// Global Variable Definitions
CGameApp    g_App;

// Name : WinMain() (Application Entry Point)
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPTSTR lpCmdLine, int iCmdShow )
{
   int retCode;

   // Initialize the engine.
   if (!g_App.InitInstance( hInstance, lpCmdLine, iCmdShow )) return 0;
```

```
    // Begin the game play process. Will return when app due to exit.
    retCode = g_App.BeginGame();

    // Shut down the engine, just to be polite, before exiting.
    if ( !g_App.ShutDown() )
     {
           MessageBox( 0, _T("Failed to shut system down correctly, please
                       check file named 'debug.txt'.\r\n\r\nIf the problem
                       persists, please contact technical support."),
                       _T("Non-Fatal Error"), MB_OK | MB_ICONEXCLAMATION );
     }

     // Return the correct exit code.
     return retCode;
}
```

First we declare a global instance of the CGameApp class. The WinMain function will call a member of the CGameApp class called InitInstance which sets up the environment. It creates the window, builds the 3D objects and allocates memory to be used as an off screen rendering target which holds the current frame (referred to as the ***frame buffer***). When the InitInstance function returns, it either returns zero, which means something has gone wrong during initialization (and we should exit immediately), or it means the application has been successfully initialized and the CGameApp class has everything it needs to start running.

The next function we call is CGameApp::BeginGame**.** This function will sit in a loop, updating the 3D scene and the screen image for each frame and handling Windows messages via the message pump. Only if a request to quit the program is found in the message queue will this function exit from the loop and return control back to the WinMain call.

Before exiting we call the CGameApp::ShutDown function. It takes care of releasing all memory used by the application. If something goes wrong during shutdown this function returns a false value and we have an opportunity to warn the user that memory may not have been released properly.

CObject

The CObject class contains the object's world matrix and a pointer to the mesh that this object will use for rendering. This demo will have two objects that share the same mesh. This shows us how instancing can be used to place multiple objects in the world while only having one physical set of mesh data in memory.

```
class CObject
{
public:
    // Constructors & Destructors for This Class.
    CObject( CMesh *pMesh );
    CObject();

    //Public Variables for This Class
    D3DXMATRIX  m_mtxWorld;            // Objects matrix
```

```
    CMesh       *m_pMesh;               // Mesh we are instancing

};
```

The default constructor initializes the CMesh pointer to NULL and sets the WorldMatrix to an identity matrix.

```
CObject::CObject()
{
    // Reset / Clear all required values
    m_pMesh = NULL;
    D3DXMatrixIdentity( &m_mtxWorld );
}
```

The second constructor allows us to attach a CMesh:

```
CObject::CObject( CMesh *pMesh )
{
    // Reset / Clear all required values
    D3DXMatrixIdentity( &m_mtxWorld );

    // Set Mesh
    m_pMesh = pMesh;
}
```

The overall picture is:

a) Each object points to a mesh and has its own World Matrix
b) Each Mesh manages an array of polygons
c) Each polygon manages an array of vertices
d) Each object will transform its mesh's vertices into world space using its world matrix

CGameApp

For the most part, the CGameApp class ***is*** the application. Here is its class definition: (see CGameApp.h)

```
class CGameApp
{
public:
    //--------------------------------------------------------------------
    // Constructors & Destructors for This Class.
    //--------------------------------------------------------------------
     CGameApp();
     virtual ~CGameApp();

    //--------------------------------------------------------------------
    // Public Functions for This Class
    //--------------------------------------------------------------------
```

```
    LRESULT     DisplayWndProc( HWND hWnd, UINT Message, WPARAM wParam,
                                LPARAM lParam );
    bool        InitInstance( HANDLE hInstance, LPCTSTR lpCmdLine,
                              int iCmdShow );
    int         BeginGame( );
    bool        ShutDown( );

 private:
    //-----------------------------------------------------------------------
    // Private Functions for This Class
    //-----------------------------------------------------------------------
    bool        BuildObjects( );
    void        FrameAdvance( );
    bool        CreateDisplay( );
    void        SetupGameState( );
    void        AnimateObjects( );
    void        PresentFrameBuffer( );
    void        ClearFrameBuffer( ULONG Color );
    bool        BuildFrameBuffer( ULONG Width, ULONG Height );
    void        DrawPrimitive( CPolygon * pPoly, D3DXMATRIX * pmtxWorld );
    void        DrawLine( const D3DXVECTOR3 & vtx1,
                          const D3DXVECTOR3 & vtx2,ULONG Color );



    //-----------------------------------------------------------------------
    // Private Static Functions For This Class
    //-----------------------------------------------------------------------
    static LRESULT CALLBACK StaticWndProc(HWND hWnd, UINT Message,
                                          WPARAM wParam, LPARAM lParam);

    //-----------------------------------------------------------------------
    // Private Variables For This Class
    //-----------------------------------------------------------------------
    D3DXMATRIX  m_mtxView;          // View Matrix
    D3DXMATRIX  m_mtxProjection;    // Projection matrix

    CMesh       m_Mesh;             // Mesh to be rendered
    CObject     m_pObject[2];       // Objects storing mesh instances

    CTimer      m_Timer;            // Game timer

    HWND        m_hWnd;             // Main window HWND
    HDC         m_hdcFrameBuffer;   // Frame Buffers Device Context
    HBITMAP     m_hbmFrameBuffer;   // Frame buffers Bitmap
    HBITMAP     m_hbmSelectOut;     // Used for selecting out of the DC

    bool        m_bRotation1;       // Object 1 rotation enabled / disabled
    bool        m_bRotation2;       // Object 2 rotation enabled / disabled

    ULONG       m_nViewX;           // X Position of render viewport
    ULONG       m_nViewY;           // Y Position of render viewport
    ULONG       m_nViewWidth;       // Width of render viewport
    ULONG       m_nViewHeight;      // Height of render viewport

};
```

D3DXMATRIX m_mtxView
The View matrix in this application is set to an identity matrix because we allow no camera movement. In the next demo, the view matrix will be used to allow you to move the camera dynamically about the scene.

D3DXMATRIX m_mtxProjection
The projection matrix that the application will use is set once at application start-up.

CMesh m_Mesh
There will be two objects in our world. Each one will use the same mesh. Therefore we only need a single mesh for this application. This mesh will be a cube and both objects will instance it.

CObject m_pObject[2]
The application will use two objects that share the same mesh. Each object has its own world matrix so it can be positioned anywhere in the 3D world. You can change the size of this array so that the application supports more objects.

CTimer m_Timer
This class allows us to get runtime reports on how the application is performing and how many frames per second are currently being rendered. It uses the high-performance counter available on most modern PCs to report very accurate timing. The timer is also used to track how much time has passed since the last frame so that we know by how much to rotate the cubes. If we did not use a timer for this, the rotation speed would not be consistent across machines. A faster machine would spin the cube more quickly as it could execute more game loops per second. This approach allows us to work with rotations specified in ***rotations per second*** and use the timer to rotate the cube for only the fraction of the second that has currently passed. We will examine this simple class later.

HWND m_hWnd
This will hold the handle to the main application window where the rendering will take place.

HDC m_hdcFrameBuffer
HBITMAP m_hbmFrameBuffer
These two member variables hold the handle to the device context (DC) and the bitmap that will be used as a frame buffer. The scene is rendered each frame to the bitmap. Once the scene has been fully drawn we will blit the bitmap to the main application window.

Note: In order to render the scene each frame, we must first erase what was drawn in the last frame. If we had a cube that was rotating between two frames and we didn't clear the old image before we rendered the newly rotated cube, we would have two sets of cube lines on the screen: one set in its old position and one set in the new position.

If we only cleared the window every frame and then drew the new scene, two undesirable effects would occur. The first problem is flicker. If we were to clear the window first to a white color and then display it and then draw the scene, you would see a flicker effect even though it happens extremely quickly (perhaps 40-60 times per second). Secondly, if you are on a low-end machine and have a low frame rate, you might actually see the scene being rendered.

Both of these conditions are unacceptable. The solution (which nearly all games implement) is an offscreen buffer used to compose the image for each frame first. This frame then replaces the old image in the previous frame and animation is achieved. This technique is referred to as **double buffering**.

So we will create a bitmap that is the same size as the portion of the application window to which we will be rendering (the application window client area). We then create a compatible DC into which we can select the bitmap. We will use the DC's drawing commands to render not to the window, but to the bitmap. When we have drawn all the lines and the image is complete, we call the DCs BitBlit function to perform a high-speed image copy from the bitmap to the application window client area. Once the bitmap is copied to the window, we can leave the user looking at the scene, while in the background the bitmap is cleared to white, thereby erasing the previously rendered lines, and then render the scene again for the next frame. We do not have to clear the application window because the bitmap copied to the window will completely cover up the previous frame that was rendered. This means we will have no flickering.

The buffer to which the scene is rendered (in our demo, the bitmap) is often called the frame buffer, because it is where we will draw the current frame to be displayed.

```
bool m_bRotation1
bool m_bRotation2
```

This application will create two cube objects and will rotate them continuously. These two boolean variables are used by the CGameApp class to toggle whether Object1 and Object2 should be rotated in each frame. Our application will have a menu that allows the user to toggle each rotation.

```
ULONG m_nViewX;
ULONG m_nViewY;
ULONG m_nViewWidth;
ULONG m_nViewHeight;
```

These four variables define the rectangle in the window to which we wish to render. For our application, these variables will store the size of the application window client area. These variables can be adjusted so that the scene is only rendered to a portion of the client area. They are the values used in mapping the 2D projection space vertices in the [–1 , +1] range to valid screen coordinates using the formula described earlier:

```
ScreenX =  Vector.x * m_nViewportWidth  / 2 + m_nViewportX + m_nViewportWidth  / 2

ScreenY = -Vector.y * m_nViewportHeight / 2 + m_nViewportY + m_nViewportHeight / 2
```

CGameApp::InitInstance

```
bool CGameApp::InitInstance( HANDLE hInstance, LPCTSTR lpCmdLine,
                             int iCmdShow )
{
    // Create the primary display device
    if (!CreateDisplay()) { ShutDown(); return false; }

    // Build Objects
    if (!BuildObjects()) { ShutDown(); return false; }

    // Set up all required game states
    SetupGameState();

    // Success!
    return true;
}
```

The first step is calling CGameApp::CreateDisplay. This function is responsible for creating and initializing the application main window. If this call fails, we return the failure so that the ***WinMain*** function can exit the application with an error.

The second function call is to CGameApp::BuildObjects. This function creates the single cube mesh and initializes both cube objects.

The final call is to CGameApp::SetupGameState which creates the application projection matrix and the view matrix. (Because we are not yet allowing camera movement, the view matrix can be initialized and left as an identity matrix).

CGameApp::CreateDisplay

The first thing we will do is create a string for our window title and use two local variables to hold the desired width and height of our window (this demo window will be 400x400).

```
bool CGameApp::CreateDisplay()
{
    LPTSTR  WindowTitle  = _T("Software Render");
```

```
    USHORT  Width        = 400;
    USHORT  Height       = 400;
    HDC     hDC          = NULL;
    RECT    rc;
```

If you are not familiar with basic Windows programming techniques then it is strongly recommended that you take the Game Institute course *Introduction to C++ Programming*. It is vital that you know how to do this.

Next we fill in our WNDCLASS structure so that we can register the type of window we wish to create with the operating system.

```
    // Register the new windows window class.
    WNDCLASS wc;
    wc.style         = CS_BYTEALIGNCLIENT | CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc   = StaticWndProc;
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = (HINSTANCE)GetModuleHandle(NULL);
    wc.hIcon         = LoadIcon( wc.hInstance,MAKEINTRESOURCE(IDI_ICON));
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH )GetStockObject(BLACK_BRUSH);
    wc.lpszMenuName  = NULL;
    wc.lpszClassName = WindowTitle;
    RegisterClass(&wc);
```

We specified a style that forces the horizontal position of the window to be byte aligned. This allows certain optimizations when we are copying the frame buffer to the window and the speed gain is quite significant. The other styles simply specify that we want Windows to repaint the window when it is resized horizontally (CS_HREDRAW) or vertically (CS_VREDRAW).

We set the icon to the one stored in the executable's resource, a standard cursor, and the background brush to black. The string 'Software Render' will be the window class name used to create an instance of the window. Note that after calling RegisterClass no window has yet been created. We have simply provided a template describing appearance and behavior.

Next we create the application window using the Win32 CreateWindow function. We pass in the window class name (this is the name we assigned when we registered the class: Basic Demo). The second parameter is the string that we would like displayed in the window caption bar (we use the same string).

```
// Create the rendering window
m_hWnd = CreateWindow( WindowTitle,
                       WindowTitle,
                       WS_OVERLAPPEDWINDOW,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       Width,  Height,
                       NULL,
                       LoadMenu( wc.hInstance,MAKEINTRESOURCE(IDR_MENU) ),
                       wc.hInstance, this );
```

```
    // Bail on error
    if (!m_hWnd) return false;
```

We ask for a 400x400 overlapped window and assign a menu to this window that gets loaded from our resource data. This menu can be viewed through the resource editor and holds commands that allow cube rotation manipulation and other directives. If the window is not created successfully, we return 'false' to the calling function.

Next, we retrieve the client area of our newly created window and assign the dimensions of the client area to our four class variables. These variables hold the rendering viewport dimensions needed for mapping the 2D projection space points to screen space.

```
   // Retrieve the final client size of the window
    ::GetClientRect( m_hWnd, &rc );
    m_nViewX      = rc.left;
    m_nViewY      = rc.top;
    m_nViewWidth  = rc.right - rc.left;
    m_nViewHeight = rc.bottom - rc.top;
```

Once our window is created, we will create the frame buffer. This is the bitmap where all rendering will take place. We then show the window, and return 'true' to indicate successful initialization.

```
    // Build the frame buffer
    if (!BuildFrameBuffer( Width, Height )) return false;


    // Show the window
    ShowWindow(m_hWnd, SW_SHOW);
    // Success!
    return true;
}
```

CGameApp::BuildFrameBuffer

We will need two things in order to render to the frame buffer. We need the frame buffer itself, which will be a bitmap, and we need a device context that we can use to draw onto the bitmap surface. The first thing we do in the following function, is retrieve a temporary device context for the application window and then (if not already created) we create a compatible device context that the frame buffer can use. We will store the handle to this device context in the CGameApp member variable m_hdcFrameBuffer.

```
bool CGameApp::BuildFrameBuffer( ULONG Width, ULONG Height )
{
    HDC hDC = ::GetDC( m_hWnd );
    if ( !m_hdcFrameBuffer ) m_hdcFrameBuffer = ::CreateCompatibleDC( hDC );
```

Next we create a bitmap that is compatible with the application window and store the returned handle to that bitmap in the CGameApp member variable m_hbmFrameBuffer. We also take care to release any previously allocated frame buffer data prior to this step, not shown here.

```
    m_hbmFrameBuffer = CreateCompatibleBitmap( hDC, Width, Height );
    if ( !m_hbmFrameBuffer ) return false;
```

We select this bitmap into the device context we created for it earlier and it is ready to be used as our frame buffer. Note that when you select an object into a device context, any previously selected object of the same type is returned from the function call. You should store this object and make sure that you select the default object back into the device context before you destroy it. For this reason we made a copy of the default bitmap returned from the SelectObject function and stored it in the CGameApp member variable m_hbmSelectOut. You should do this with any objects that you intend to select into a device context, including pens and brushes. If you fail to restore a device context to its default state before releasing it, your application (as well as any other applications running concurrently) may not perform properly until the operating system is rebooted. On earlier versions of Windows this is especially true; device contexts were a very limited resource.

```
    m_hbmSelectOut = (HBITMAP)::SelectObject( m_hdcFrameBuffer,
                                              m_hbmFrameBuffer );
```

Finally we release the window DC (because we only used it to create a compatible DC for the bitmap) and set the frame buffer DC so that it renders transparently.

```
    ::ReleaseDC( m_hWnd, hDC );
    ::SetBkMode( m_hdcFrameBuffer, TRANSPARENT );
    return true;
}
```

CGameApp::BuildObjects()

The CGameApp class has a single mesh which will hold our cube. We call the Mesh's ***AddPolygon*** function to add the 6 faces of our cube.

```
bool CGameApp::BuildObjects()
{
    CPolygon *pPoly = NULL;
    if ( m_Mesh.AddPolygon( 6 ) < 0 ) return false;
```

For each polygon we now add four vertices that define the model space coordinates of the corner points of that face. This is similar to the cube example code we looked at earlier in this lesson:

```
    // Front Face
    pPoly = m_Mesh.m_pPolygon[0];
    if ( pPoly->AddVertex( 4 ) < 0 ) return false;
```

```
    pPoly->m_pVertex[0] = CVertex( -2,  2, -2 );
    pPoly->m_pVertex[1] = CVertex(  2,  2, -2 );
    pPoly->m_pVertex[2] = CVertex(  2, -2, -2 );
    pPoly->m_pVertex[3] = CVertex( -2, -2, -2 );

    // Top Face
    pPoly = m_Mesh.m_pPolygon[1];
    if ( pPoly->AddVertex( 4 ) < 0 ) return false;

    pPoly->m_pVertex[0] = CVertex( -2,  2,  2 );
    pPoly->m_pVertex[1] = CVertex(  2,  2,  2 );
    pPoly->m_pVertex[2] = CVertex(  2,  2, -2 );
    pPoly->m_pVertex[3] = CVertex( -2,  2, -2 );

    // Back Face
    pPoly = m_Mesh.m_pPolygon[2];
    if ( pPoly->AddVertex( 4 ) < 0 ) return false;

    pPoly->m_pVertex[0] = CVertex( -2, -2,  2 );
    pPoly->m_pVertex[1] = CVertex(  2, -2,  2 );
    pPoly->m_pVertex[2] = CVertex(  2,  2,  2 );
    pPoly->m_pVertex[3] = CVertex( -2,  2,  2 ),

    // Bottom Face
    pPoly = m_Mesh.m_pPolygon[3];
    if ( pPoly->AddVertex( 4 ) < 0 ) return false;

    pPoly->m_pVertex[0] = CVertex( -2, -2, -2 );
    pPoly->m_pVertex[1] = CVertex(  2, -2, -2 );
    pPoly->m_pVertex[2] = CVertex(  2, -2,  2 );
    pPoly->m_pVertex[3] = CVertex( -2, -2,  2 );

    // Left Face
    pPoly = m_Mesh.m_pPolygon[4];
    if ( pPoly->AddVertex( 4 ) < 0 ) return false;

    pPoly->m_pVertex[0] = CVertex( -2,  2,  2 );
    pPoly->m_pVertex[1] = CVertex( -2,  2, -2 );
    pPoly->m_pVertex[2] = CVertex( -2, -2, -2 );
    pPoly->m_pVertex[3] = CVertex( -2, -2,  2 );

    // Right Face
    pPoly = m_Mesh.m_pPolygon[5];
    if ( pPoly->AddVertex( 4 ) < 0 ) return false;

    pPoly->m_pVertex[0] = CVertex(  2,  2, -2 );
    pPoly->m_pVertex[1] = CVertex(  2,  2,  2 );
    pPoly->m_pVertex[2] = CVertex(  2, -2,  2 );
    pPoly->m_pVertex[3] = CVertex(  2, -2, -2 );
```

We now have our mesh created and all polygons defined. Next we need to assign this single mesh to both objects in our game world. This is a classic example of instancing mesh data. Our world will contain two objects, but only one mesh will be used by both:

```
    // Our two objects should reference this mesh
    m_pObject[ 0 ].m_pMesh = &m_Mesh;
    m_pObject[ 1 ].m_pMesh = &m_Mesh;
```

Finally we set each object world matrix so that they are positioned in different locations in world space. Object0 will be centered at world space vector (-3.5, 2, 14) and Object1 will be positioned at world space vector (3.5, -2, 14).

```
    // Set both objects matrices so that they are offset slightly
    D3DXMatrixTranslation(&m_pObject[0].m_mtxWorld, -3.5f,  2.0f, 14.0f );
    D3DXMatrixTranslation(&m_pObject[1].m_mtxWorld,  3.5f, -2.0f, 14.0f );

    // Success!
    return true;
}
```

Because we are setting the view matrix to identity, our camera will be located at world space position (0, 0, 0) with a look vector of (0, 0, 1). This means that both cubes will be located at a distance of 14 units in front of the camera. Both will be offset horizontally and vertically from the camera 3.5 units and 2.0 units respectively in opposing directions.

Notice that we use the D3DX library to build our translation matrix for each object. Both cubes will initially not be rotated with regards to the world space axes.

CGameApp::SetupGameState

```
void CGameApp::SetupGameState()
{
    float      fAspect;
    D3DXMatrixIdentity( &m_mtxView );
```

The first thing this function does is set the application view matrix to an identity matrix. Remember that in this demo we are not going to be manipulating the view matrix. Thus we can set it up once at application start-up and forget about it. Remember that an identity matrix provides no translation values and will align objects with the standard world axes. Again, this is equivalent to us explicitly placing our camera at world space coordinate (0,0,0) looking down the positive Z axis with a look vector of (0,0,1) and an up vector of (0,1,0).

Identity View Matrix

Right Vector

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

←----------Translation Vector (0,0,0)----------→

In later demo applications we will manipulate the view matrix to allow us to move the camera about the 3D world. When we do this we will have to rebuild the view matrix every time the camera position or rotation changes.

Our next task is to build the projection matrix using the D3DXMatrixPerspectiveFovLH function. In order to avoid image distortion when mapping from the projection window to the viewport, we calculate the aspect ratio of the viewport and pass it into the function. Here we are asking for a projection matrix that gives us a vertical FOV of 60 degrees (D3DXToRadian is a helper function that automatically converts degrees to radians).

```
    fAspect = (float)m_nViewWidth / (float)m_nViewHeight;
    D3DXMatrixPerspectiveFovLH( &m_mtxProjection, D3DXToRadian( 60.0f ),
                                fAspect, 1.01f, 1000.0f );
```

The last two parameters to the above function can be ignored for now as they are used for clipping and depth buffer coordinate mapping which are not used in this application. The resulting matrix is stored in the CGameApp::m_mtxProjection member variable. Finally, we set both objects to a true rotation status:

```
    // Enable rotation for both objects
    m_bRotation1 = true;
    m_bRotation2 = true;
}
```

CGameApp::BeginGame

When InitInstance returns, we call the CGameApp::BeginGame function. This is the function that will contain the main message processing and render loop. It will not return program flow back to WinMain until the user chooses to close the application. This is very similar to how MFC encapsulates the message pump within the CWinApp::Run function.

```
int CGameApp::BeginGame()
{
    MSG msg;

    // Start main loop
    while (1)
    {
        // Did we receive a message, or are we idling ?
        if ( PeekMessage(&msg, NULL, 0, 0, PM_REMOVE) )
        {
                if (msg.message == WM_QUIT) break;

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

```
        }

    } // Until quit message is received

    return 0;
}
```

The BeginGame function sits in a loop calling CGameApp::FrameAdvance in order to redraw the scene. Before rendering we check to make sure that there are no messages in the message pump which need to be handled. If there are messages, we need to remove them and send them off to the OS for processing. These messages will eventually be routed to our ***StaticWndProc*** function that handles the application level processing of these messages.

The only message we are interested in for now is the WM_QUIT message. It tells us that the user has attempted to close the application. We will need to break out of our infinite loop and return back to our WinMain function, where the function will end and the application will be shut down.

CGameApp::FrameAdvance

The FrameAdvance function is called repeatedly by the BeginGame function. It will apply rotations to the objects, clear the frame buffer to erase the previous frame, render each of the objects to the frame buffer and copy the contents of the frame buffer to the main application window whereby a new frame is displayed to the user.

```
void CGameApp::FrameAdvance()
{
   CMesh      *pMesh = NULL;
   TCHAR       lpszFPS[30];
```

The first thing we do is advance the timer since we need to keep track of the time that has passed between the previous frame and the current one. The CTimer::Tick function retrieves the current time from the high performance counter and updates its internal variables so that we can access the data later on. The parameter passed in is a frame rate ceiling value. This locks the frame rate to prevent it from updating too quickly on very fast computers. In our demo we use a value of 60. This means we desire to update the screen no more than 60 times per second. The CTimer::Tick function will burn up any extra time to make this so:

```
   m_Timer.Tick( 60.0f );
```

Next we call the CGameApp::AnimateObjects function. This is the function that applies the rotations to the object world matrices.

```
   AnimateObjects();
```

Then we call CGameApp::ClearFrameBuffer to erase the previous frame image from our frame buffer bitmap. It uses the frame buffer device context to draw a large rectangle over the entire bitmap. The color of the rectangle is the value passed into this function (in our demo, bright white). This allows us to have any background color we want on the frame buffer. Be sure to check the source code for implementation details.

```
    ClearFrameBuffer( 0x00FFFFFF );
```

Having our clean frame buffer and all rotations applied to our object world matrices, we are ready to draw those objects in their newly rotated positions. We now begin our render loop. For each object we get a pointer to its mesh and then loop through each of the polygons. For each polygon we call the CGameApp::DrawPrimitive function which will take care of rendering the wire frame polygon to the frame buffer.

```
  for ( ULONG i = 0; i < NumberOfObjects; i++ )
  {
    pMesh = m_pObject[i].m_pMesh;

    // Loop through each polygon
    for ( ULONG f = 0; f < pMesh->m_nPolygonCount; f++ )
    {
      DrawPrimitive( pMesh->m_pPolygon[f], &m_pObject[i].m_mtxWorld );
    }
  }
```

When the above code exits, all objects have had their polygons rendered into the frame buffer. The scene is now ready to be displayed to the user. However, before we do that we call CTimer::GetFrameRate and pass it a string to fill with frame rate information. This string is also added to the frame buffer:

```
  m_Timer.GetFrameRate( lpszFPS );
  TextOut( m_hdcFrameBuffer, 5, 5, lpszFPS, strlen( lpszFPS ) );
```

Finally we present the newly rendered frame to the user. The CGameApp::PresentFrameBuffer call performs the copying of the frame buffer bitmap to the application window client area.

```
    PresentFrameBuffer();
}
```

CGameApp::AnimateObjects

This function creates the rotation matrices which are later multiplied by each object world matrix to create a new world matrix which has been rotated from its previous position. This can be done using fewer lines of code then we will see below (and we will examine a shorter version later). The reason we have expanded this code is that it better demonstrates the matrix multiplication process.

First we create some local D3DXMATRIX variables to hold Yaw, Pitch and Roll data. Another matrix (mtxRotate) will hold the concatenated result of multiplying these matrices. We also use three local float variables that will be used to hold the appropriate angles.

```
void CGameApp::AnimateObjects()
{
    D3DXMATRIX mtxYaw, mtxPitch, mtxRoll, mtxRotate;
    float RotationYaw, RotationPitch, RotationRoll;
```

If the user has not disabled the rotation of Object1 then we create some rotational values. These are arbitrary values and can be modified. We selected a yaw rotation value of 75 degrees per second, a pitch rotation value of 50 degrees per second and a roll value of 25 degrees per second. Multiplying these values by the fraction of a second returned from the CTimer::GetTimeElapsed function scales them accordingly. If we are running at, say, 4 frames per second, this call would return 0.25 which will scale the yaw rotation value to 18.75. This allows for rotation to be independent of frame rate.

```
    // Rotate Object 1 by small amount
    if ( m_bRotation1 )
    {
        RotationYaw   = D3DXToRadian( 75.0f * m_Timer.GetTimeElapsed() );
        RotationPitch = D3DXToRadian( 50.0f * m_Timer.GetTimeElapsed() );
        RotationRoll  = D3DXToRadian( 25.0f * m_Timer.GetTimeElapsed() );
```

Using these values you can see that the object rotates around the X axis at twice the rate it rotates about the Z axis, and rotates about the Y axis three times the amount it rotates about the Z axis.

With our yaw, pitch and roll rotation values we build three rotation matrices. We also create an identity matrix to hold the concatenation of all three matrices.

```
        // Build rotation matrices
        D3DXMatrixIdentity( &mtxRotate );
        D3DXMatrixRotationY( &mtxYaw, RotationYaw);
        D3DXMatrixRotationX( &mtxPitch,RotationPitch);
        D3DXMatrixRotationZ( &mtxRoll, RotationRoll);
```

The next step is to use the D3DXMatrixMultiply (which multiplies two matrices) function to combine all of these rotations into a final matrix. This function is an alternative to using the overloaded * operator. We use D3DXMatrixMultiply to better see the multiplication order.

```
        // Concatenate the rotation matrices
        D3DXMatrixMultiply( &mtxRotate, &mtxRotate, &mtxYaw );
        D3DXMatrixMultiply( &mtxRotate, &mtxRotate, &mtxPitch );
        D3DXMatrixMultiply( &mtxRotate, &mtxRotate, &mtxRoll );
```

The resulting matrix is returned to us in the mtxRotate variable. It contains all of the rotations for the x, y and z axes that need to be applied to the first object. All that is left to do is multiply this matrix with the object's current world matrix and we are done:

```
        D3DXMatrixMultiply( &m_pObject[ 0 ].m_mtxWorld, &mtxRotate,
                            &m_pObject[ 0 ].m_mtxWorld );
 } // End if Rotation Enabled
```

Object1 now has its world matrix updated to contain the new rotations. When this matrix is used to transform the mesh vertices later, the object will be rendered in its new orientation.

We repeat the same steps for Object2 and the function returns.

For completeness, here is some code that could be used to make the function smaller:

```
void CGameApp::AnimateObjects()
{
    D3DXMATRIX mtxYaw, mtxPitch, mtxRoll, mtxRotate;
    float RotationYaw, RotationPitch, RotationRoll;



    if ( m_bRotation1 )
    {
        RotationYaw   = D3DXToRadian( 75.0f * m_Timer.GetTimeElapsed() );
        RotationPitch = D3DXToRadian( 50.0f * m_Timer.GetTimeElapsed() );
        RotationRoll  = D3DXToRadian( 25.0f * m_Timer.GetTimeElapsed() );

        // Build entire rotation matrix
        D3DXMatrixRotationYawPitchRoll(&mtxRotate , RotationYaw , RotationPitch,
                                       RotationRoll);

        // Multiply with world matrix using operators
        m_pObject[0].m_mtxWorld = mtxRotate * m_pObject[0].m_mtxWorld;

    } // End if Rotation Enabled

    if ( m_bRotation2 )
    {
        RotationYaw   = D3DXToRadian( -25.0f * m_Timer.GetTimeElapsed() );
        RotationPitch = D3DXToRadian(  50.0f * m_Timer.GetTimeElapsed() );
        RotationRoll  = D3DXToRadian( -75.0f * m_Timer.GetTimeElapsed() );

        // Build entire rotation matrix
        D3DXMatrixRotationYawPitchRoll(&mtxRotate , RotationYaw , RotationPitch,
                                       RotationRoll);

       // Multiply with world matrix using operators
       m_pObject[1].m_mtxWorld = mtxRotate * m_pObject[1].m_mtxWorld;

    } // End if rotation enabled
}
```

As you can see we have used D3DXMatrixRotationYawPitchRoll to build a matrix that contains all three rotations in one call. The resulting matrix is multiplied with the object world matrices using the overloaded * operator instead of the D3DXMatrixMultiply function.

CGameApp::DrawPrimitive

The CGameApp::DrawPrimitive function renders our polygons. It is this function that is responsible for transforming the polygons from model space to screen space and then drawing them to the frame buffer. This is the heart of our rendering pipeline.

```
void CGameApp::DrawPrimitive( CPolygon * pPoly, D3DXMATRIX * pmtxWorld )
{
    D3DXVECTOR3 vtxPrevious, vtxCurrent;

    // Loop round each vertex transforming as we go
    for ( USHORT v = 0; v < pPoly->m_nVertexCount + 1; v++ )
    {
        // Store the current vertex
        vtxCurrent = (D3DXVECTOR3&)pPoly->m_pVertex[ v % pPoly->m_nVertexCount ];
```

First we loop through each vertex in the polygon and store the current vertex in the vtxCurrent vector. The ***[v % pPoly->m_nVertexCount]*** line makes certain that we wrap around to vertex zero again for the end point of the last line. You will notice that we loop + 1 times more than there are vertices in the polygon. This is because a final line will be drawn between the last vertex and vertex zero.

The object that this polygon belongs to has had its world matrix passed in so we can multiply each vertex with this matrix to transform it into world space:

```
        // Multiply the vertex position by the World / object matrix
        D3DXVec3TransformCoord( &vtxCurrent, &vtxCurrent, pmtxWorld );
```

The vertex is now in world space and is ready to be transformed into view space.

```
        // Multiply by View matrix
        D3DXVec3TransformCoord( &vtxCurrent, &vtxCurrent, &m_mtxView );
```

The vertex is now in view space. We will now multiply it with the projection matrix so that it can be deformed (squashed or expanded) to simulate the requested FOV.

```
  // Multiply by Projection matrix
  D3DXVec3TransformCoord( &vtxCurrent, &vtxCurrent, &m_mtxProjection );
```

The D3DXVec3TransformCoord function does an automatic divide by w to ensure that a 3D vector is returned. In the previous two function calls this has had no effect as both the world matrix and the view matrix have identity W columns (therefore w=1 in the resulting vector). This is not so with our projection matrix. The W column of this matrix is set up so that the input vector's Z component is copied into the output vector's W component. Since this function performs the homogenization before it returns the vector, it will not only multiply the vector by the projection matrix, but it will also divide the x,y,z components by w (performing the 2D projection). So when this function returns, the 3D vector has x and y components in 2D projection space (z can be ignored for now) where valid

coordinates are in the –1 to +1 range. Our final transformation converts the 2D homogeneous clip space coordinates into screen coordinates using the formula covered in this lesson's textbook:

```
 // Convert to screen space coordinates
 vtxCurrent.x = vtxCurrent.x * m_nViewWidth/2 + m_nViewX + m_nViewWidth/ 2;
 vtxCurrent.y =-vtxCurrent.y * m_nViewHeight/2 + m_nViewY + m_nViewHeight/2;
```

We now have our vertex in screen space such that the x and y components of the 3D vector are relative to the pixel in the top left corner of our window.

If this is the first vertex of the polygon we are transforming, we will skip the DrawLine function and store this vertex in the vtxPrevious local variable. Each time through this loop we will draw the line from the previous vertex to the vertex that has just been transformed:

```
 // If this is the first vertex, continue. This is the first
 // point of our first line.
 if ( v == 0 ) { vtxPrevious = vtxCurrent; continue; }

   // Draw the line
   DrawLine( vtxPrevious, vtxCurrent, 0 );

   // Store this as new line's first point
   vtxPrevious = vtxCurrent;

 } // Next Vertex
}
```

Here is the complete function:

```
void CGameApp::DrawPrimitive( CPolygon * pPoly, D3DXMATRIX * pmtxWorld )
{
 D3DXVECTOR3 vtxPrevious, vtxCurrent;
 for ( USHORT v = 0; v < pPoly->m_nVertexCount + 1; v++ )
 {
   vtxCurrent = (D3DXVECTOR3&)pPoly->m_pVertex[ v % pPoly->m_nVertexCount ];

   D3DXVec3TransformCoord( &vtxCurrent, &vtxCurrent, pmtxWorld );
   D3DXVec3TransformCoord( &vtxCurrent, &vtxCurrent, &m_mtxView );
   D3DXVec3TransformCoord( &vtxCurrent, &vtxCurrent, &m_mtxProjection );

   vtxCurrent.x =   vtxCurrent.x * m_nViewWidth  / 2 + m_nViewX + m_nViewWidth  / 2;
   vtxCurrent.y =  -vtxCurrent.y * m_nViewHeight / 2 + m_nViewY + m_nViewHeight / 2;

   if ( v == 0 ) { vtxPrevious = vtxCurrent; continue; }

   DrawLine( vtxPrevious, vtxCurrent, 0 );

   vtxPrevious = vtxCurrent;

 }
}
```

CGameApp::DrawLine

In this function we create a black pen, select it into the frame buffer device context, and then render the line between the two screen space vectors using the LineTo function. Notice that although the vectors passed are 3D vectors, the z component is unused and the x and y components are in screen space.

```
void CGameApp::DrawLine(const D3DXVECTOR3 &vtx1, const D3DXVECTOR3 &vtx2,
                        ULONG Color )
{
    LOGPEN logPen;
    HPEN   hPen = NULL, hOldPen = NULL;


// Set up the rendering pen
    logPen.lopnStyle   = PS_SOLID;
    logPen.lopnWidth.x = 1;
    logPen.lopnWidth.y = 1;

    // Set up the color, converted to BGR & stripped of alpha
    logPen.lopnColor   = 0x00FFFFFF & RGB2BGR( Color );

    // Create the rendering pen
    hPen = ::CreatePenIndirect( &logPen );
    if (!hPen) return;

    // Select into the frame buffer DC
    hOldPen = (HPEN)::SelectObject( m_hdcFrameBuffer, hPen );

    // Draw the line segment
    MoveToEx( m_hdcFrameBuffer, (long)vtx1.x, (long)vtx1.y, NULL );
    LineTo( m_hdcFrameBuffer, (long)vtx2.x, (long)vtx2.y );

    // Destroy rendering pen
    ::SelectObject( m_hdcFrameBuffer, hOldPen );
    ::DeleteObject( hPen );
}
```

CGameApp::PresentFrameBuffer

PresentFrameBuffer retrieves the device context of the application window and calls the Win32 BitBlt function to copy the image from the frame buffer device context to the application window device context. After this, the application window device context is released as it is no longer needed. As mentioned earlier, device contexts are valuable resources and should be released back to the operating system whenever they are not needed.

```
void CGameApp::PresentFrameBuffer( )
{
    HDC hDC = NULL;

    // Retrieve the DC of the window
    hDC = ::GetDC(m_hWnd);
```

```
    // Blit the frame buffer to the screen
    ::BitBlt( hDC, m_nViewX, m_nViewY, m_nViewWidth, m_nViewHeight,
              m_hdcFrameBuffer, m_nViewX, m_nViewY, SRCCOPY );

    // Clean up
    ::ReleaseDC( m_hWnd, hDC );
}
```

CGameApp::StaticWndProc

When you register a window class under the Windows operating system you must specify a function through which Windows will route all messages that were received by that window. This callback function can handle requests from the user dealing with keyboard and mouse input, as well as menu selections and application window closing. In order for this to work, Windows is very specific about how the function should be declared. It must have the following definition:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT Message, WPARAM wParam, LPARAM lParam);
```

Many developers use global functions for this purpose but there are alternatives. It is often preferable to map these messages to a member function instead (in our CGameApp class for example). However, when we call a class member function in C++, the compiler inserts an invisible first parameter to the parameter list (a ***this*** pointer). The 'this' pointer points to the current instance of the class so that all class instances can share the same physical function code. This presents a problem since the parameters passed into the function no longer match up with the function signature that Windows requires. A static member function will solve this problem.

When we create a static member function for a class, the function acts just like a global function and has no *this* pointer, but is confined to the class namespace. Even if no instances of the CGameApp class have been created, we can still call the CGameApp:StaticWndProc function because the function is created at application start-up just like a global function and is shared by all instances of the CGameApp class. Recall that when using such functions the only class member variables that are accessible are those declared as static as well. This is logical since accessing a non-static member variable when no class instances have been created would be difficult (since those variables have not been constructed yet). Static class member variables are like static class member functions in that they are shared by all instances of the class and are created at application start-up just like global variables. This means that they can be accessed and assigned values even if no instances of the class have been created. They must be accessed using the class namespace:

```
CGameApp::MyVariable = 1;
```

Our CGameApp class does not use static member variables but it does use a static member function to distribute the window messages to the correct instance of the class. Please note that while we only ever have one instance of the CGameApp class in this demo, it does allow us flexibility in the future to have several CGameApp classes running in a single application, as well as being able to work directly within our game application object.

To begin, let us recall how we created the window:

```
m_hWnd = CreateWindow(  WindowTitle, WindowTitle, WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, CW_USEDEFAULT, Width, Height, NULL,
                        LoadMenu( wc.hInstance, MAKEINTRESOURCE(IDR_MENU) ),
                        wc.hInstance, this );
```

Notice that the last parameter to be passed in, is the '**this'** pointer. It maps to a pointer to the instance of the CGameApp class that created the window. This allows us to pass application-defined data to the window procedure when it is created.

Once the window has been created our window procedure receives a WM_CREATE message. The lparam parameter in the WndProc function will point to a CreateStruct. The CreateStruct contains all the creation information about our window. More importantly, it has a field in the structure called CREATESTRUCT::lpCreateParams which contains the application-defined data which was sent in as the last parameter in the CreateWindow call. In our case, this information will be a pointer to the instance of the application class that created the window. This is important because our static window function is shared by all class instances. It will need to know for which instance of the CGameApp class this message is intended.

In Windows, every window has a 4 byte user data area where you can store application-defined data to be associated with the window. In the following code you can see that we use the Win32 SetWindowLong function to store the CGameApp pointer passed to the function in the user data area. This means that the window itself now stores the instance of the CGameApp for which it was created.

```
LRESULT CALLBACK CGameApp::StaticWndProc(HWND hWnd, UINT Message, WPARAM
                                         wParam, LPARAM lParam)
{
    if ( Message == WM_CREATE ) SetWindowLong( hWnd,
                                               GWL_USERDATA,
                                              (LONG)((CREATESTRUCT FAR*)
                                               lParam)->lpCreateParams);
```

This happens only once when the window is created and the WM_CREATE message is received. It is important if you use this method yourself to make sure that you pass in a pointer to the instance of the class in the call to CreateWindow

We are somewhat limited for any other messages because we cannot access any of the member variables of the class. This is because we are in a static function. So what we will do instead is retrieve the CGameApp pointer from the window that sent the message using the Win32 GetWindowLong function. Once we have this pointer we have the instance of the CGameApp for which the message is intended.

```
    CGameApp *Destination = (CGameApp*)GetWindowLong( hWnd, GWL_USERDATA );
```

We can now forward the message to one of CGameApp's non-static member functions. DisplayWindowProc handles windows messages for our application.

```
    if (Destination)
        return Destination->DisplayWndProc( hWnd, Message, wParam, lParam );
```

If we receive a message that has not yet had a pointer to an instance of CGameApp assigned to it, we will forward this message to the OS for default message processing.

```
    // No destination found, defer to system...
    return DefWindowProc( hWnd, Message, wParam, lParam );
}
```

CGameApp::DisplayWndProc

This function handles messages for the application object. It checks for menu items being selected and requests to close the application. It also traps the WM_SIZE message so that if the window is being resized the projection matrix can be rebuilt to take into account the new aspect ratio of the viewport dimensions.

```
LRESULT CGameApp::DisplayWndProc( HWND hWnd, UINT Message, WPARAM wParam,
                                  LPARAM lParam )
{
    float fAspect;

    // Determine message type
    switch (Message)
    {
         case WM_CREATE:
         break;
         case WM_CLOSE:
                 PostQuitMessage(0);
         break;
         case WM_DESTROY:
                 PostQuitMessage(0);
         break;
```

We do not want any action taken in the case of a WM_CREATE message since we have already handled it in the parent function described previously. In the case of the application being closed by the user or the application window being explicitly destroyed, we call the Win32 PostQuitMessage function. This will send a WM_QUIT message to the application. The WM_QUIT message is polled for in CGameApp::BeginGame and used to break from the infinite rendering loop.

One of the messages that we must be on the lookout for is the WM_SIZE message (sent when the user has resized the application window). This directly affects our rendering since it alters the aspect ratio of the rendering window. This means that we will need to recalculate the aspect ratio using the new window dimensions and rebuild the projection matrix to take these dimensions into account. Also note that the frame buffer is a bitmap and it has to match the dimensions of the window as well. The BuildFrameBuffer function has already been covered and takes care of destroying any previously created frame buffer.

```
        case WM_SIZE:

           // Store new viewport sizes
           m_nViewWidth  = LOWORD( lParam );
           m_nViewHeight = HIWORD( lParam );

           // Set up new perspective projection matrix
           fAspect = (float)m_nViewWidth / (float)m_nViewHeight;
           D3DXMatrixPerspectiveFovLH( &m_mtxProjection,
                                       D3DXToRadian( 60.0f ),
                                       fAspect, 1.01f, 1000.0f );

           // Rebuild the new frame buffer
           BuildFrameBuffer( m_nViewWidth, m_nViewHeight );
        break;
```

We exit the application in response to the user pressing the escape key, so we must trap the WM_KEYDOWN message and check the wParam variable passed in to see what key was pressed:

```
        case WM_KEYDOWN:
         // Which key was pressed?
           switch (wParam)
           {
                   case VK_ESCAPE:
                       PostQuitMessage(0);
                       return 0;
           }
```

The last section of code traps command messages generated by the user selecting a menu item. These simple menu items allow the user to toggle the state of an object's rotation variable. We also check to see if the user has selected the ***Exit*** command from the menu.

```
case WM_COMMAND:

     // Process Menu Items
     switch( LOWORD(wParam) )
     {
      case ID_ANIM_ROTATION1:
           // Disable / enable rotation
           m_bRotation1 = !m_bRotation1;
           ::CheckMenuItem( ::GetMenu( m_hWnd ), ID_ANIM_ROTATION1,
                                    MF_BYCOMMAND | (m_bRotation1) ?
                                    MF_CHECKED :  MF_UNCHECKED );
           break;

      case ID_ANIM_ROTATION2:
           // Disable / enable rotation
           m_bRotation2 = !m_bRotation2;
           ::CheckMenuItem( ::GetMenu( m_hWnd ), ID_ANIM_ROTATION2,
                                    MF_BYCOMMAND | (m_bRotation2) ?
                                    MF_CHECKED :  MF_UNCHECKED );
           break;
      case ID_EXIT:
           // Received key/menu command to exit app
           SendMessage( m_hWnd, WM_CLOSE, 0, 0 ); return 0;
```

```
      } // End Switch

      default:
         return DefWindowProc(hWnd, Message, wParam, lParam);

    } // End Message Switch

    return 0;
}
```

Any messages that we do not handle directly are passed on to Windows for default processing by calling the Win32 function DefWindowProc.

Exercises

The current demo does not allow the camera to be moved and the view matrix is left as an identity matrix. In this exercise, try adding user input to the demo so that the user can strafe the camera left or right in response to the left and right cursor keys being pressed.

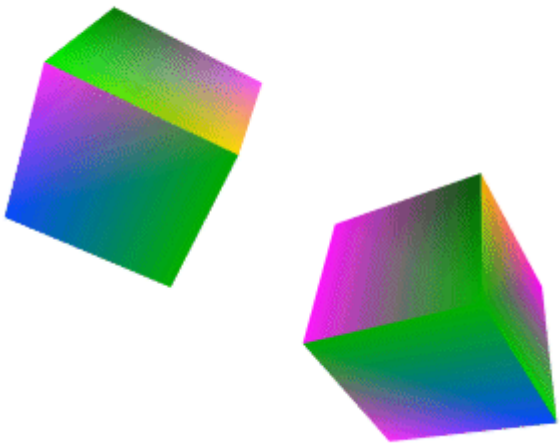
Tips:

a) You will need to add key handlers in the DisplayWndProc function .
b) You can set the view matrix to an identity matrix at application start up as we have done in our code, but you will need to modify the view matrix in response to the left or right keys being pressed. It is the translation information (the last row) of the view matrix that will have to be updated.
c) In order to strafe the camera, you will need to move the camera along its RIGHT VECTOR. Refer back to the diagram of the view matrix to see how to extract the right vector. (hint: look at column 1)
d) Remember that multiplying the right vector with a negative distance value will move the camera left.
e) Remember to store the newly translated vector back into the translation row of the view matrix before rendering the scene.

Further Reading:

The CTimer class used by our demo uses the Windows high performance counter functions. We have provided a short document explaining how to use the timer function. This document can be found accompanying this material (in the download section), and is named TimerTut.zip

Workbook Chapter Two: DirectX Graphics Foundation



Lab Project 2.1 Device Initialization

The first demo in this chapter creates a device in windowed mode. We will not have to concern ourselves yet with the enumeration of all possible fullscreen modes since we will simply use the mode currently in use by the desktop. The initialization demo is almost identical in structure to the software demo we implemented in the last lesson. We have kept all of the same function names and the same CGameApp class. It is probably a good idea to open up the project to follow along with the explanations. Focus on the DirectX Graphics code that creates the initial Direct3D9 object and then creates a valid device.

CGameApp::CreateDisplay

In our last demo, WinMain called the CGameApp::InitInstance function which in turn called the CGameApp::CreateDisplay function to create the main application window. In this demo, we add a new function called CGameApp::InitDirect3D which is called before the function exits:

```
bool CGameApp::CreateDisplay()
{
    LPTSTR WindowTitle   = _T("Initialization");
    USHORT Width         = 400;
    USHORT Height        = 400;
    RECT   rc;

    // Register the new windows window class.
    WNDCLASS wc;
    wc.style                  = CS_BYTEALIGNCLIENT | CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc            = StaticWndProc;
    wc.cbClsExtra             = 0;
    wc.cbWndExtra             = 0;
    wc.hInstance              = (HINSTANCE)GetModuleHandle(NULL);
    wc.hIcon                  = LoadIcon( wc.hInstance, MAKEINTRESOURCE(IDI_ICON));
    wc.hCursor                = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground          = (HBRUSH )GetStockObject(BLACK_BRUSH);
    wc.lpszMenuName           = NULL;
    wc.lpszClassName          = WindowTitle;
    RegisterClass(&wc);

    // Create the rendering window
    m_hWnd = CreateWindow( WindowTitle, WindowTitle, WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                           CW_USEDEFAULT, Width, Height, NULL,
                           LoadMenu( wc.hInstance, MAKEINTRESOURCE(IDR_MENU) ),
                           wc.hInstance, this );

    // Bail on error
    if (!m_hWnd) return false;

    // Retrieve the final client size of the window
    ::GetClientRect( m_hWnd, &rc );
    m_nViewX        = rc.left;
    m_nViewY        = rc.top;
    m_nViewWidth    = rc.right - rc.left;
```

```
    m_nViewHeight   = rc.bottom - rc.top;

    // Show the window
    ShowWindow(m_hWnd, SW_SHOW);

    // Initialize Direct3D (Simple)
    if (!InitDirect3D( )) return false;

    // Success!!
    return true;
}
```

Notice that we called the function that is responsible for initializing the DirectX Graphics environment after we created the application window. This is important since we will need the window handle to create a valid Direct3DDevice9 object. The device needs to know where its frame buffer will ultimately be copied each frame. You can think of the client area of the window created above as being the front buffer.

Our CGameApp class will have two new members:

```
LPDIRECT3D9             m_pD3D;          // Direct3D Object
LPDIRECT3DDEVICE9       m_pD3DDevice;    // Direct3D Device Object
```

These pointers are used to store the IDirect3D9 interface and the IDirect3DDevice9 interface that will be returned to us after we create the respective objects.

CGameApp::InitDirect3D

The CGameApp class has member variable pointers to the IDirect3D9 interface (m_pD3D) and the IDirect3DDevice9 interface. These are the interfaces that will be created in this function if it is successful. CGameApp also has a member variable of type `D3DPRESENT_PARAMETERS` (m_D3DPresentParams) that will contain the presentation parameters used to create the device. This will be useful if we need to rebuild the device at a later stage.

```
bool CGameApp::InitDirect3D()
{
    D3DPRESENT_PARAMETERS PresentParams;
    D3DCAPS9              Caps;
    D3DDISPLAYMODE        CurrentMode;
    HRESULT               hRet;

    // First of all create our D3D Object
    m_pD3D = Direct3DCreate9( D3D_SDK_VERSION );
    if (!m_pD3D)  return false;
```

First, we create some local variables to store intermediate information. The D3DDISPLAYMODE structure will be used to obtain and store the display mode currently being used by the primary adapter to display the Windows desktop.

Then, we attempt to create the Direct3D9 object. If successful, the returned pointer to an IDirect3D9 interface is stored in the CGameApp class member variable m_pD3D. If the call is unsuccessful and m_pD3D is NULL, then something is terribly wrong and the application cannot continue. This is likely the result of incorrect (or non-existent) installation of DirectX.

Now, we will fill out the D3DPRESENT_PARAMETERS structure in preparation for passing it to the IDirect3D9::CreateDevice function. For the most part, environment setup is no more complicated than figuring out the correct values to store in this structure and then passing it into the CreateDevice function. Just to be safe, the first thing we do is zero out the structure.

```
    // Fill out a simple set of present parameters
   ZeroMemory( &PresentParams, sizeof(D3DPRESENT_PARAMETERS) );
```

Next, we use the IDirect3D9::GetAdapterMode function and pass it the address of a D3DISPLAYMODE. The function will fill this structure with the current adapter display mode. We have specified the D3DADAPTER_DEFAULT flag which means that we are asking for the current display mode of the primary adapter.

We store the pixel format of the returned display mode in the BackBufferFormat member of the D3DPRESENT_PAREMETERS structure. This informs the device that we want a frame buffer with a matching pixel format.

```
    // Select back buffer format etc
   m_pD3D->GetAdapterDisplayMode( D3DADAPTER_DEFAULT, &CurrentMode);
   PresentParams.BackBufferFormat = CurrentMode.Format;
```

Next, we set the EnableAutoDepthStencil member to TRUE indicating our desire for a depth buffer that is attached to the frame buffer at device creation time. We also set the depth buffer pixel format. To do so, we will call a helper function (that we will write) called FindDepthStencilFormat. It will return a valid D3DFORMAT that works with this device.

```
    //Setup remaining flags
   PresentParams.EnableAutoDepthStencil = true;
   PresentParams.AutoDepthStencilFormat =
                FindDepthStencilFormat( D3DADAPTER_DEFAULT, CurrentMode, D3DDEVTYPE_HAL );
```

The next member we fill in is the SwapEffect. We use D3DSWAPEFFECT_DISCARD to allow the device to choose the presentation approach. We also set the Windowed member of the structure to TRUE so that we can run the application in windowed mode.

```
   PresentParams.SwapEffect     = D3DSWAPEFFECT_DISCARD;
   PresentParams.Windowed       = true;
```

The remaining presentation parameters can be left at zero. The device will choose the appropriate default behaviors for these members as discussed in the text.

The next task is to determine whether the HAL device on this system supports hardware vertex processing and choose the optimal approach.

```
    // Set Creation Flags
    unsigned long ulFlags = D3DCREATE_SOFTWARE_VERTEXPROCESSING;

    // Check if Hardware T&L is available
    ZeroMemory(&Caps, sizeof(D3DCAPS9));
    m_pD3D->GetDeviceCaps( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, &Caps );

    if ( Caps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT )
                               ulFlags = D3DCREATE_HARDWARE_VERTEXPROCESSING;
```

The IDirect3D9::GetDeviceCaps above returns a D3DCAPS9 structure. It contains the functionality and capabilities of a particular device type on a particular adapter. This structure is quite large and you can examine its members in the SDK documentation. We will cover many of its members throughout this course as our application hardware requirements grow and checking for capabilities becomes more important. In the code we are asking for the capabilities structure for the HAL device on the primary adapter. If the HAL device supports transformation and lighting in hardware as well as rasterization, the DevCaps member of the D3DCAPS9 structure will have the D3DDEVCAPS_HWTRANSFORMANDLIGHT flag set. If this flag is not set then either the HAL only supports rasterization (in which case we must create the device using the D3DCREATE_SOFTWARE_VERTEXPROCESSING flag) or a HAL device is not present on the hardware. We will now try to create a device. We pass the adapter ordinal we wish to create the device for (D3DADPATER_DEFAULT), the device type we wish to create (a HAL device), the HWND of the focus window (created in the CreateDisplay function and also used as the device window by default), the vertex processing we wish the device to use, the address of the presentation parameters structure that we filled in above, and the address of a pointer to an IDirect3DDevice9 interface that will be filled in if the call is successful.

```
    // Attempt to create a HAL device
    if( FAILED( hRet = m_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, m_hWnd,
                                             ulFlags, &PresentParams, &m_pD3DDevice ) ) )
    {
```

If device creation fails, we will try to create a HEL device (reference rasterizer). If this is the case, we should amend the AutoDepthStencilFormat field with a depth surface pixel format that is compatible with the REF device. We call our helper function again to test depth buffer formats for the reference rasterizer and return the best supported format:

```
        // Find REF depth buffer format
        PresentParams.AutoDepthStencilFormat =
                  FindDepthStencilFormat( D3DADAPTER_DEFAULT, CurrentMode, D3DDEVTYPE_REF );

         // Check if Hardware T&L is available
         ZeroMemory(&Caps , sizeof(D3DCAPS9));
         ulFlags = D3DCREATE_SOFTWARE_VERTEXPROCESSING;
         m_pD3D->GetDeviceCaps( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, &Caps );
         if ( Caps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT )
```

```
                    ulFlags = D3DCREATE_HARDWARE_VERTEXPROCESSING;

        // Attempt to create a REF device
        if( FAILED( hRet = m_pD3D->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, m_hWnd,
                                                ulFlags, &PresentParams, &m_pD3DDevice ) ) )
        {
            // Failed
            return false;

        } // End if Failure (REF)

    } // End if Failure (HAL)
```

If the reference device could not be created, then something is terribly wrong and we will have no choice but to exit the application with an error. If our device is successfully created, then we store the presentation parameters so that we can access them later if need be.

```
    // Store the present parameters
    m_D3DPresentParams = PresentParams;

    // Success!!
    return true;
}
```

You might find it odd that after the HAL device creation failed, we did not immediately create the REF device with software vertex processing. After all, it is a software device with no hardware capabilities available. While this is true, recall that the purpose of the reference rasterizer is to emulate a hardware device. When we create certain resources in DirectX Graphics, how we create them depends on whether or not we are using hardware or software vertex processing. Therefore, if you coded your application so that it only worked with hardware vertex processing and did not have hardware vertex processing capabilities on your development machine, you could create a reference rasterizer with hardware vertex processing and you would not have to change all of your resource creation function calls. Of course, when you create a reference device with hardware vertex processing, nothing is processed in hardware. But since it pretends that it is, your application can treat it in exactly the same way as a HAL device and keep the same resource creation code. Here is the function in its entirety for easier reading:

```
bool CGameApp::InitDirect3D()
{
    HRESULT               hRet;
    D3DPRESENT_PARAMETERS PresentParams;
    D3DCAPS9              Caps;
    D3DDISPLAYMODE        CurrentMode;

    // First of all create our D3D Object
    m_pD3D = Direct3DCreate9( D3D_SDK_VERSION );
    if (!m_pD3D)     return false;

     // Fill out a simple set of present parameters
    ZeroMemory( &PresentParams, sizeof(D3DPRESENT_PARAMETERS) );

    // Select back buffer format etc
```

```
    m_pD3D->GetAdapterDisplayMode( D3DADAPTER_DEFAULT, &CurrentMode);
    PresentParams.BackBufferFormat = CurrentMode.Format;

    //Setup remaining flags
    PresentParams.EnableAutoDepthStencil = true;
    PresentParams.AutoDepthStencilFormat =
                FindDepthStencilFormat( D3DADAPTER_DEFAULT, CurrentMode, D3DDEVTYPE_HAL );
    PresentParams.SwapEffect  = D3DSWAPEFFECT_DISCARD;
    PresentParams.Windowed    = true;

    // Set Creation Flags
    unsigned long ulFlags = D3DCREATE_SOFTWARE_VERTEXPROCESSING;

    // Check if Hardware T&L is available
    ZeroMemory(&Caps , sizeof(D3DCAPS9));
    m_pD3D->GetDeviceCaps( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, &Caps );

    if ( Caps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT )
          ulFlags = D3DCREATE_HARDWARE_VERTEXPROCESSING;

    // Attempt to create a HAL device
    if( FAILED( hRet = m_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, m_hWnd,
                                             ulFlags, &PresentParams, &m_pD3DDevice ) ) )
    {
        // Find REF depth buffer format
        PresentParams.AutoDepthStencilFormat =
                 FindDepthStencilFormat( D3DADAPTER_DEFAULT, CurrentMode, D3DDEVTYPE_REF );

        // Check if Hardware T&L is available
        ZeroMemory(&Caps , sizeof(D3DCAPS9));
        ulFlags = D3DCREATE_SOFTWARE_VERTEXPROCESSING;
        m_pD3D->GetDeviceCaps( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, &Caps );

        if ( Caps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT )
           ulFlags = D3DCREATE_HARDWARE_VERTEXPROCESSING;

        // Attempt to create a REF device
        if( FAILED( hRet = m_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF,
                                                 m_hWnd, ulFlags, &PresentParams,
                                                 &m_pD3DDevice ) ) )
        {
            // Failed
            return false;
        } // End if Failure (REF)

    } // End if Failure (HAL)

    // Store the present parameters
    m_D3DPresentParams = PresentParams;

    // Success!!
    return true;
}
```

CGameApp::InitInstance

```
bool CGameApp::InitInstance( HANDLE hInstance, LPCTSTR lpCmdLine, int iCmdShow )
{
    // Create the primary display device
    if (!CreateDisplay()) { ShutDown(); return false; }

    // Build Objects
    if (!BuildObjects())   { ShutDown(); return false; }

    // Set up all required game states
    SetupGameState();

    // Setup our rendering environment
    SetupRenderStates();

    // Success!
    return true;
}
```

So far we have covered only the CreateDisplay function. This function created the application window and then called the InitDirect3D function to create the device object. The next function we need to look at is the BuildObjects function which (as with our Chapter 1 demo) creates the mesh used by both cube objects.

Our vertex class will now include a color in addition to position data. The class includes a new constructor which takes X, Y and Z position components along with a DWORD describing the color.

```
class CVertex
{
public:
    // Public Variables for This Class
    float       x;          // Vertex Position X Component
    float       y;          // Vertex Position Y Component
    float       z;          // Vertex Position Z Component
    ULONG   Diffuse;        // Diffuse Vertex Color Component

    // Constructors & Destructors for This Class.
    CVertex( float fX, float fY, float fZ, ULONG ulDiffuse = 0xFF000000 )
    {
            x = fX;
            y = fY;
            z = fZ;
            Diffuse = ulDiffuse;
    }

    CVertex()     { x = 0.0f; y = 0.0f; z = 0.0f; Diffuse = 0xFF000000; }
};
```

The CVertex class definition can be found in CObject.h. If no color is specified, a default color of black is used. Remember that colors are in ARGB format, so this default color is A=255, R=0, G=0, B=0. An alpha value of 255 indicates a solid (opaque) color. We will discuss alpha components later in the course.

Apart from these few changes, there is virtually no difference between this demo and the last demo in the way that objects and meshes are stored. Each object in the world contains a pointer to a mesh and a world matrix describing its position and orientation in the world:

```
class CObject
{
public:
    //-----------------------------------------------------------------------
    // Constructors & Destructors for This Class.
    //-----------------------------------------------------------------------
    CObject( CMesh * pMesh );
    CObject();

   //-----------------------------------------------------------------------
   // Public Variables for This Class
   //-----------------------------------------------------------------------
    D3DXMATRIX  m_mtxWorld;       // Objects world matrix
    CMesh      *m_pMesh;          // Mesh we are instancing
};
```

The CMesh and CPolygon classes are identical to the classes used last time so they are not shown again here.

CGameApp::BuildObjects

The CGameApp::BuildObjects function is also identical to our last demo with the exception of the new vertex class. We create a 4x4 cube mesh and assign it to both of our world objects. We set each object world matrix to an arbitrary position that looks good for the demo.

We used a macro called RANDOM_COLOR (main.h) to generate a random color to send into each vertex constructor. At the beginning of the function, we call **srand** to seed the random number generator so that we get different random numbers generated each time the application is run. We do this by seeding with the current time. timeGetTime returns a DWORD value describing the amount of time elapsed (in milliseconds) since Windows was started. This function wraps around to zero again every 2^32 milliseconds (about every 49.5 days).

```
#define RANDOM_COLOR 0xFF000000 | ((rand() * 0xFFFFFF) / RAND_MAX)

bool CGameApp::BuildObjects()
{
    CPolygon *pPoly = NULL;

    // Seed the random number generator
    srand( timeGetTime() );

    // Add 6 polygons to this mesh.
    if ( m_Mesh.AddPolygon( 6 ) < 0 ) return false;

    // Front Face
    pPoly = m_Mesh.m_pPolygon[0];
```

```
    if ( pPoly->AddVertex( 4 ) < 0 ) return false;

    pPoly->m_pVertex[0] = CVertex( -2,  2, -2, RANDOM_COLOR );
    pPoly->m_pVertex[1] = CVertex(  2,  2, -2, RANDOM_COLOR );
    pPoly->m_pVertex[2] = CVertex(  2, -2, -2, RANDOM_COLOR );
    pPoly->m_pVertex[3] = CVertex( -2, -2, -2, RANDOM_COLOR );

    // Top Face
    pPoly = m_Mesh.m_pPolygon[1];
    if ( pPoly->AddVertex( 4 ) < 0 ) return false;

    pPoly->m_pVertex[0] = CVertex( -2,  2,  2, RANDOM_COLOR );
    pPoly->m_pVertex[1] = CVertex(  2,  2,  2, RANDOM_COLOR );
    pPoly->m_pVertex[2] = CVertex(  2,  2, -2, RANDOM_COLOR );
    pPoly->m_pVertex[3] = CVertex( -2,  2, -2, RANDOM_COLOR );

    // Back Face
    pPoly = m_Mesh.m_pPolygon[2];
    if ( pPoly->AddVertex( 4 ) < 0 ) return false;

    pPoly->m_pVertex[0] = CVertex( -2, -2,  2, RANDOM_COLOR );
    pPoly->m_pVertex[1] = CVertex(  2, -2,  2, RANDOM_COLOR );
    pPoly->m_pVertex[2] = CVertex(  2,  2,  2, RANDOM_COLOR );
    pPoly->m_pVertex[3] = CVertex( -2,  2,  2, RANDOM_COLOR );

    // Bottom Face
    pPoly = m_Mesh.m_pPolygon[3];
    if ( pPoly->AddVertex( 4 ) < 0 ) return false;

    pPoly->m_pVertex[0] = CVertex( -2, -2, -2, RANDOM_COLOR );
    pPoly->m_pVertex[1] = CVertex(  2, -2, -2, RANDOM_COLOR );
    pPoly->m_pVertex[2] = CVertex(  2, -2,  2, RANDOM_COLOR );
    pPoly->m_pVertex[3] = CVertex( -2, -2,  2, RANDOM_COLOR );

    // Left Face
    pPoly = m_Mesh.m_pPolygon[4];
    if ( pPoly->AddVertex( 4 ) < 0 ) return false;

    pPoly->m_pVertex[0] = CVertex( -2,  2,  2, RANDOM_COLOR );
    pPoly->m_pVertex[1] = CVertex( -2,  2, -2, RANDOM_COLOR );
    pPoly->m_pVertex[2] = CVertex( -2, -2, -2, RANDOM_COLOR );
    pPoly->m_pVertex[3] = CVertex( -2, -2,  2, RANDOM_COLOR );

    // Right Face
    pPoly = m_Mesh.m_pPolygon[5];
    if ( pPoly->AddVertex( 4 ) < 0 ) return false;

    pPoly->m_pVertex[0] = CVertex(  2,  2, -2, RANDOM_COLOR );
    pPoly->m_pVertex[1] = CVertex(  2,  2,  2, RANDOM_COLOR );
    pPoly->m_pVertex[2] = CVertex(  2, -2,  2, RANDOM_COLOR );
    pPoly->m_pVertex[3] = CVertex(  2, -2, -2, RANDOM_COLOR );

    // Our two objects should reference this mesh
    m_pObject[ 0 ].m_pMesh = &m_Mesh;
    m_pObject[ 1 ].m_pMesh = &m_Mesh;

    // Set both objects matrices so that they are offset slightly
    D3DXMatrixTranslation( &m_pObject[ 0 ].m_mtxWorld, -3.5f,  2.0f, 14.0f );
    D3DXMatrixTranslation( &m_pObject[ 1 ].m_mtxWorld,  3.5f, -2.0f, 14.0f );
```

```
    // Success!
    return true;
}
```

Notice that each cube face has only four vertices stored in a clockwise order and no duplicated are used within the face. But we do have duplicated vertices *between* faces. Every corner point will have three vertices in identical positions belonging to different faces. This is unavoidable but not necessarily undesirable because it allows us to provide each vertex in every face a unique color. Fig 2.1 shows the output from this first demo:

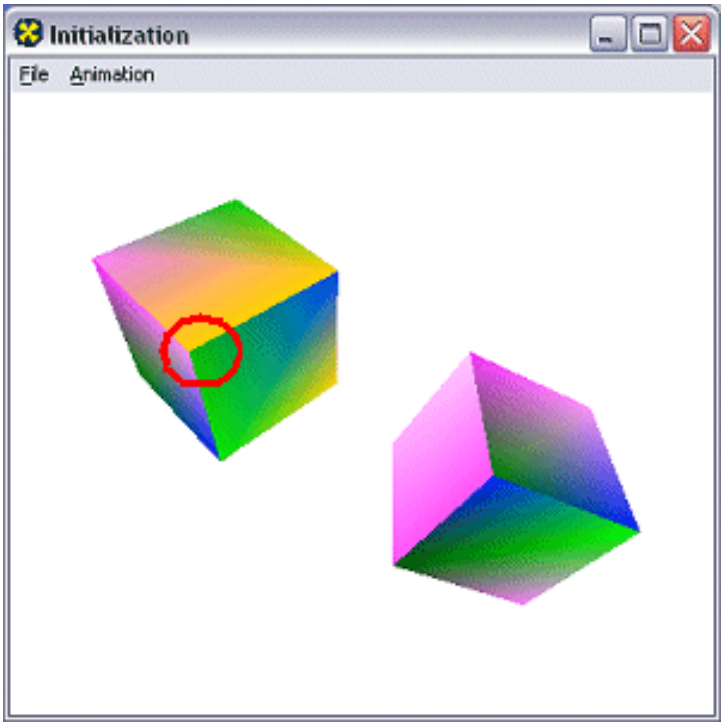


Figure 2.1

We note that the random colors stored at each of the face vertices are smoothly interpolated across the face, blending from one color to the next. This is because we are using Gouraud shading. The highlighted corner position is shared by three faces, but is a different color in each face. This is why the faces each need a copy of their own vertex at that position; the top face has a yellow vertex, the right face has a green vertex and the left face has a pink vertex. If we disabled Gouraud shading, each face would be the color of the color of its first vertex.

CGameApp::SetupGameState

Once control is returned to CGameApp::InitInstance, the next function called is SetupGameState to set any states needed within the application itself. We create an identity matrix which will be used later to initialize the device view matrix. We set the m_bRotation boolean to true for each object since we want both cubes to be animated:

```
void CGameApp::SetupGameState()
{
    // Setup Default Matrix Values
    D3DXMatrixIdentity( &m_mtxView );

    // Enable rotation
    m_bRotation1 = true;
    m_bRotation2 = true;

    // App is active
    m_bActive    = true;
}
```

CGameApp::SetupRenderStates

SetupRenderStates is the last function called by InitInstance in our application framework. Its job is to initialize our projection matrix, vertex format, transform states and render states prior to entering the main rendering loop.

```
void CGameApp::SetupRenderStates()
{
    // Set up new perspective projection matrix
    float fAspect = (float)m_nViewWidth / (float)m_nViewHeight;
    D3DXMatrixPerspectiveFovLH( &m_mtxProjection, D3DXToRadian( 60.0f ), fAspect,
                                1.0f , 1000.0f  );

    // Setup our D3D Device initial states
    m_pD3DDevice->SetRenderState( D3DRS_ZENABLE, D3DZB_TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_DITHERENABLE,  TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_SHADEMODE, D3DSHADE_GOURAUD );
    m_pD3DDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CCW );
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, FALSE );

    // Setup our vertex FVF flags
    m_pD3DDevice->SetFVF( D3DFVF_XYZ | D3DFVF_DIFFUSE );

    // Setup our matrices
    m_pD3DDevice->SetTransform( D3DTS_VIEW, &m_mtxView );
    m_pD3DDevice->SetTransform( D3DTS_PROJECTION, &m_mtxProjection );
}
```

Note the last two parameters in the D3DX projection matrix creation function. These were values that we deliberately avoided discussing in Chapter 1 because we did not have the background at that point

to understand what they represented. The next section examines these values (Near and Far planes) and their purpose.

Near and Far Planes

In Chapter 1 we examined the means by which the first and second columns of the projection matrix scale vertices so that arbitrary fields of view can be achieved. We also discussed how those columns integrated the concept of the aspect ratio to adjust the horizontal FOV to compensate for display windows (especially in full screen) that have more pixels in the X direction than they do in the Y direction. We looked at setting up the fourth column (W column) as a Z identity column so that the W component in the output vector was equal to the view space Z component in the input vector. Finally we divided the X and Y components of the output vector by the W component of the output vector to provide perspective projection onto the projection window.

The third column of the matrix was not discussed at the time because our assertion was that it was only useful if a depth buffer was being used (which we did not use in our software renderer). It is now time to revisit this third column since we intend to use a Z-Buffer from this point forward.

In DirectX Graphics, we must set up the third column of the projection matrix in such a way that we ensure that when the output vector Z component is divided by W, the resulting depth values are in the [0.0, 1.0] range.

This may seem trivial at first glance, but recall that the divide by W is dividing the output vector components by the Z component of the input vector (because output W = input Z). So the input Z component must be altered in some way or we will end up with an output vector from the projection matrix where:

V(x,y,z,1) * ProjectionMatrix = H(X , Z , Z=z , W=z)

Because the W and Z components are the same, we would always end up with a depth for that vertex of 1.0. This is hardly useful. Our goal is to ensure that the output vector Z component is different than the output vector W component such that dividing Z by W provides a value in the range of 0.0 to 1.0.

This means that the view space Z values input into the projection matrix multiplication must be mapped to some other range. The solution is to define a minimum and maximum range of acceptable Z values.

For this, the concept of a far plane is used. A **far plane** is a plane set at some application defined maximum distance from the camera. That distance defines a maximum value for view space Z components that can be considered for rendering. For example, let us say that we decide that we want a far plane at a distance of 400 units from the camera. Vertices that are on the opposite side of this far plane will be clipped and discarded. This provides a finite range of Z values and makes a mapping possible:

Source Range	Target Range
0 – 400	0.0 – 1.0

In order to avoid rendering artifacts for objects that are too close to the camera (or even behind it), a **near plane** is constructed. This sets a minimum distance value for vertices in relation to the camera. The near plane must be some finite distance from the camera (it cannot have a distance of zero). Developers typically use a near plane distance between 1.0 and 10.0. Trial and error is often required to find a nice distance for the application. Triangles or vertices closer to the camera than the near plane distance are clipped and discarded from the render list.

The last two parameters in the D3DXMatrixPerspeciveLH call are the view space distance for the near and far planes. In the above code we have used a near plane value of 1.0 and a far plane value of 10000. Recall in chapter 1 that the projection matrix created a view cone by setting up the FOV in the X and Y dimensions. If we include these planes in the overall picture, the result is a new shape which we call a **frustum**. A frustum looks like a pyramid with a flat top instead of a point:

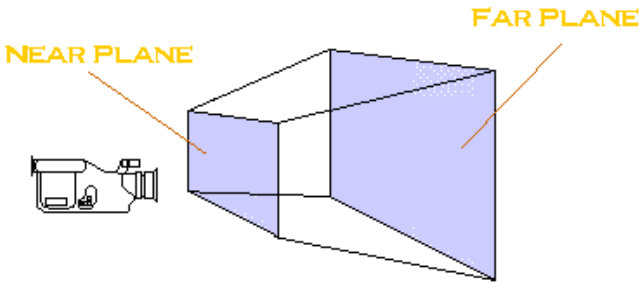


Figure 2.2

This truncated pyramid shape is characterized by six planes (near, far, top, bottom, left, right). Vertices that do not fall within the six planes created by the projection matrix are clipped. In other words, only triangles that are not completely outside the frustum are rendered. Triangles that are partially inside the frustum are clipped by DirectX Graphics so that only the fragment of the triangle that was originally inside the frustum is sent on to the rasterizer. Since D3DXMatrixPerspectiveLH handles building the third column of the projection matrix using these near and far plane values, we are spared the need to do so ourselves. However, if you wish to understand how these values are integrated into the matrix, please read Appendix A at the end of this lesson.

The appendix will also discuss why this mapping from one range to another is responsible for the Z-Buffer having a non-linear mapping (which can lead to certain artifacts – especially with 16 bit Z-Buffers).

The Render Loop

Application setup is complete and we are ready to enter the main render loop. CGameApp::InitInstance has returned control back to the WinMain function which calls CGameApp::BeginGame class to start running the render loop. Although this is unchanged from the application code in Chapter 1, it is included below to refresh your memory. The function processes windows messages if any are available and calls CGameApp::FrameAdvance to process the next frame to be rendered.

```
int CGameApp::BeginGame()
{
    MSG msg;
    while (1)
    {
        // Did we receive a message, or are we idling ?
       if ( PeekMessage(&msg, NULL, 0, 0, PM_REMOVE) )
       {
            if (msg.message == WM_QUIT) break;

            TranslateMessage( &msg );
            DispatchMessage ( &msg );
       }
       else
       {
            // Advance Game Frame.
            FrameAdvance();

       } // End If messages waiting
    } // Until quit message is received

    return 0;
}
```

CGameApp::FrameAdvance

The bulk of the application processing is handled in the FrameAdvance function. It addresses scene animation and rendering and includes a new helper function called ProcessInput to test for key presses. This will give the user the ability to strafe the camera left and right.

```
void CGameApp::FrameAdvance()
{
    CMesh      *pMesh = NULL;

    // Advance the timer
    m_Timer.Tick( );

    // Skip if app is inactive
    if ( !m_bActive ) return;
```

We update the timer class by calling its Tick() method. This updates its internal variables with the new time. We then check whether or not the CGameApp::m_bActive variable is true. If it is not true, then

the application is currently minimized and we exit the function to free up processing time for other applications the user may be running.

```
    // Recover lost device if required
    if ( m_bLostDevice )
    {
        // Can we reset the device yet ?
        HRESULT hRet = m_pD3DDevice->TestCooperativeLevel();
        if ( hRet == D3DERR_DEVICENOTRESET )
        {
            // Restore the device
            m_pD3DDevice->Reset( &m_D3DPresentParams );
            SetupRenderStates( );
            m_bLostDevice = false;

        } // End if Can Reset
        else
        {
            // device cannot be reset at this time
            return;
        }
    } // end if Device Lost
```

To deal with the occurrence of a lost device, the application polls the current device state using the TestCooperativeLevel function. If the device cannot be reset, the function exits. This allows us to continue to poll the device on each subsequent frame until DirectX is ready to reset it. Once ready to reset the device, we pass in our stored presentation parameters and reset our flag that indicates a lost device. This takes the application out of recovery mode and allows FrameAdvance to continue its processing:

```
    // Poll & Process input devices
    ProcessInput();

    // Animate the two objects
    AnimateObjects();
```

We will examine the ProcessInput function in greater detail in a later section. At a high level, it checks for key presses that indicate the user's intention to move the camera to a new location. The function rebuilds the view matrix using this new positional information and sends it to the device for the next render.

The AnimateObjects function has not changed from our last demonstration. It applies rotations to the cube objects by adjusting the values in their world matrices.

Before we render the scene, the first thing we do is clear the frame buffer and reset the Z-Buffer. We can accomplish both of these objectives using the Clear function call via the IDirect3DDevice9 interface:

```
    // Clear the frame & depth buffer ready for drawing
    m_pD3DDevice->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, 0xFFFFFFFF, 1.0f, 0);
```

Scene rendering then starts with a call to BeginScene and then proceeds much like our last application. We loop through each of the objects in our scene (2 objects in this case), get a pointer to the object mesh and then set the device world matrix to the current object world matrix. We then proceed to loop through each polygon in the mesh and call DrawPrimitiveUP for each face. Notice that we render the two triangles of the face using the triangle fan primitive type. This allows us to pass our four face vertices in a clockwise order and have them automatically rendered as two triangles. The device will transform the input vertices using its state matrices, producing 2D screen vertices that will be used to render filled polygons. The call to EndScene informs the device that the application has finished rendering the current frame.

```
    // Begin Scene Rendering
    m_pD3DDevice->BeginScene();

    // Loop through each object
    for ( ULONG i = 0; i < 2; i++ )
    {
        // Store mesh for easy access
        pMesh = m_pObject[i].m_pMesh;

        // Set our object matrix
        m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_pObject[i].m_mtxWorld );

        // Loop through each polygon
        for ( ULONG f = 0; f < pMesh->m_nPolygonCount; f++ )
        {
            CPolygon * pPolygon = pMesh->m_pPolygon[f];

            // Render the primitive
            m_pD3DDevice->DrawPrimitiveUP(D3DPT_TRIANGLEFAN,
                                          pPolygon->m_nVertexCount - 2,
                                          pPolygon->m_pVertex, sizeof(CVertex) );

        } // Next Polygon

    } // Next Object

    // End Scene Rendering
    m_pD3DDevice->EndScene();
```

At this point, our cubes are rendered into the frame buffer. The final step is to instruct the device object to present the frame buffer to the front buffer using the IDirect3DDevice9::Present function. Notice that the Present function takes place outside of the BeginScene / EndScene pair.

```
    // Present the buffer
    if ( FAILED(m_pD3DDevice->Present( NULL, NULL, NULL, NULL )) )
        m_bLostDevice = true;
}
```

The complete frame advance function is shown next without interruption:

```
void CGameApp::FrameAdvance()
{
```

```
    CMesh      *pMesh = NULL;

    // Advance the timer
    m_Timer.Tick( );

    // Skip if app is inactive
    if ( !m_bActive ) return;

    // Recover lost device if required
    if ( m_bLostDevice )
    {
        // Can we reset the device yet ?
        HRESULT hRet = m_pD3DDevice->TestCooperativeLevel();
        if ( hRet == D3DERR_DEVICENOTRESET )
        {
            // Restore the device
            m_pD3DDevice->Reset( &m_D3DPresentParams );
            SetupRenderStates( );
            m_bLostDevice = false;

        } // End if Can Reset
        else
        {
            // device cannot be reset at this time
            return;
        }

    } // End if Device Lost

    // Poll & Process input devices
    ProcessInput();

    // Animate the two objects
    AnimateObjects();

    // Clear the frame & depth buffer ready for drawing
   m_pD3DDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, 0xFFFFFFFF, 1.0f, 0 );

    // Begin Scene Rendering
    m_pD3DDevice->BeginScene();

    // Loop through each object
    for ( ULONG i = 0; i < 2; i++ )
    {
        // Store mesh for easy access
        pMesh = m_pObject[i].m_pMesh;

        // Set our object matrix
        m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_pObject[i].m_mtxWorld );

        // Loop through each polygon
        for ( ULONG f = 0; f < pMesh->m_nPolygonCount; f++ )
        {
            CPolygon * pPolygon = pMesh->m_pPolygon[f];

            // Render the primitive
            m_pD3DDevice->DrawPrimitiveUP( D3DPT_TRIANGLEFAN, pPolygon->m_nVertexCount - 2,
                                           pPolygon->m_pVertex, sizeof(CVertex) );
```

```
        } // Next Polygon

    } // Next Object

    // End Scene Rendering
    m_pD3DDevice->EndScene();

    // Present the buffer
    if ( FAILED(m_pD3DDevice->Present( NULL, NULL, NULL, NULL )) ) m_bLostDevice = true;

}
```

CGameApp::ProcessInput

The CGameApp::ProcessInput function moves the camera left or right depending on input from the user. This is done by manipulating the current view matrix and setting it as the device view matrix for use in the transformation pipeline during the next rendering of the scene:

```
void CGameApp::ProcessInput( )
{
    // Simple strafing
    if ( GetKeyState( VK_LEFT  ) & 0xFF00 )
       m_mtxView._41 += 25.0f * m_Timer.GetTimeElapsed();
    if ( GetKeyState( VK_RIGHT ) & 0xFF00 )
       m_mtxView._41 -= 25.0f  * m_Timer.GetTimeElapsed();

    // Update the device matrix
    if (m_pD3DDevice) m_pD3DDevice->SetTransform( D3DTS_VIEW, &m_mtxView );
}
```

Our demo camera will always look straight down the positive Z axis (look vector = <0, 0, 1>). We can create the illusion of strafing (i.e. moving from side to side) by adding or subtracting offsets to the camera X position in world space. Recall that the bottom row of the view matrix is responsible for the camera position. Specifically we know that the first column of the fourth row holds the X position of the camera (actually it is an inverse X position as discussed in Chapter 1).

The m_Timer.GetTimeElapsed function returns the number of seconds that have elapsed since the previous frame. We then multiply this value by the (arbitrary) value 25. This means that the camera will slide left or right at a speed of 25 world units per second. If the application was running at 60 fps, this would move our camera a distance of 25 * (1/60) = 0.416 units per frame.

If you attempted the homework assignment at the end of Chapter 1, you now have a function with which to compare your work.

Before moving on with the rest of this lesson, it would be worthwhile for you to put this chapter aside and spend some time playing with the code before moving on. It is very important that you understand the key topics discussed because they will be fundamental to using DirectX as we move forward. Once

you feel comfortable with the information learned so far, you will be ready to move on and review Lab Project 2.2 source code.

The second demo will be identical to this first one except this time it will use an enumeration class to allow the user to select display mode settings. The enumeration class presented in the next section is fairly large. Fortunately, it can be written once and then reused in all of your future applications as a black box initialization module.

Lab Project 2.2: Device Enumeration

Lab Project 2.1 had very easy initialization code. This is mostly a result of the fact that the application ran only in windowed mode. When creating a device in windowed mode we generally use the current desktop display setting to create our device. We know these settings are supported by the adapter because the adapter is currently using them to display the desktop. The following code shows the basic initialization code from Lab Project 2.1 to refresh your memory. We fill out the **D3DPRESENT_PARAMETERS** structure with our device creation parameters and call CreateDevice as shown.

```
 D3DPRESENT_PARAMETERS PresentParams;

 // Select back buffer format etc
m_pD3D->GetAdapterDisplayMode( D3DADAPTER_DEFAULT, &CurrentMode);
PresentParams.BackBufferFormat = CurrentMode.Format;

 // Setup remaining flags
PresentParams.EnableAutoDepthStencil = true;
PresentParams.AutoDepthStencilFormat =
               FindDepthStencilFormat( D3DADAPTER_DEFAULT, CurrentMode, D3DDEVTYPE_HAL );

PresentParams.SwapEffect        = D3DSWAPEFFECT_DISCARD;
PresentParams.Windowed          = true;

 // Set Creation Flags
 unsigned long ulFlags = D3DCREATE_SOFTWARE_VERTEXPROCESSING;

 // Check if Hardware T&L is available
ZeroMemory(&Caps , sizeof(D3DCAPS9));
m_pD3D->GetDeviceCaps( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, &Caps );

 if ( Caps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT )
            ulFlags = D3DCREATE_HARDWARE_VERTEXPROCESSING;

 // Attempt to create a HAL device
 m_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, m_hWnd, ulFlags,
                       &PresentParams, &m_pD3DDevice ) );
```

Creating a fullscreen device is actually not that much more difficult to do although it will require more care. We would need to set the windowed member of the D3DPRESENT_PARAMETERS structure to false and we would fill in the BackBufferWidth and BackBufferHeight members appropriately. The

next example shows the creation of a fullscreen device running in a resolution of 800x600 using the pixel format currently used by the desktop:

```
D3DPRESENT_PARAMETERS PresentParams;
// Select back buffer format etc
m_pD3D->GetAdapterDisplayMode( D3DADAPTER_DEFAULT, &CurrentMode);
PresentParams.BackBufferFormat = CurrentMode.Format;

//Setup remaining flags
PresentParams.BackBufferWidth  = 800;
PresentParams.BackBufferHeight = 600;
PresentParams.EnableAutoDepthStencil = true;
PresentParams.AutoDepthStencilFormat =
             FindDepthStencilFormat( D3DADAPTER_DEFAULT, CurrentMode, D3DDEVTYPE_HAL );

PresentParams.SwapEffect       = D3DSWAPEFFECT_DISCARD;
PresentParams.Windowed         = false;

// Set Creation Flags
unsigned long ulFlags = D3DCREATE_SOFTWARE_VERTEXPROCESSING;

// Check if Hardware T&L is available
ZeroMemory(&Caps , sizeof(D3DCAPS9));
m_pD3D->GetDeviceCaps( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, &Caps );

if ( Caps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT )
      ulFlags = D3DCREATE_HARDWARE_VERTEXPROCESSING;

// Attempt to create a HAL device
 m_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, m_hWnd, ulFlags,
                       &PresentParams, &m_pD3DDevice ) );
```

While this would be a fairly straightforward code update, it is generally not acceptable to restrict the user to pre-selected video modes. In a commercial game, your application should provide the user with a choice of which resolution and color depth to use. These settings allow the user to tailor the performance of their computer to the application so that everything runs as smoothly as possible. Incidentally, there is no guarantee that the current display mode being used by the desktop is supported by the adapter in fullscreen mode (although this is usually the case).

Providing the user the ability to configure the application environment is not a difficult process but it is somewhat involved. The following list demonstrates typical tasks that need to be accomplished as part of a more user-friendly (and system safe) initialization process.

The application should determine:

- the number of adapters on the current system
- the display modes each adapter supports
- whether or not the adapter can create a HAL device
- hardware vertex processing for the HAL device on a given adapter
- available depth buffer formats for a given display mode/device/adapter set
- which adapter display modes work with each device type

- available refresh rates (if we wish to allow the user to select)
- available presentation intervals (if we wish to allow the user to select)
- which adapters, devices and video modes support anti-aliasing

We would also like to give the application the ability to force restrictions on device feature requirements. For example, if our application requires support for the use of a stencil buffer and the user hardware does not meet those needs, the application will need to be able to take appropriate steps when this is determined.

The CD3DInitialize Class

In this project we are going to create a Direct3D environment initialization class. It will use a number of support classes and we will be discussing those over the coming sections. Before we examine the source code to this class, it will be helpful to see how it would be used by an application. The class design embodies a simple goal: to create a valid (and preferably optimal) device. The CD3DInitialize class is defined in CD3DInitialize.h and CD3DInitialize.cpp.

The main purpose of this class is to create a valid device and return a pointer to that device to the application. It is designed to be called from an application initialization function. The class will store a great deal of information about the system environment including every combination of Adapter/Device/Display Mode/Z-Buffer formats. In our application, we will instantiate this class on the stack since this allows the information to be flushed from memory as soon as the class goes out of scope (when the initialization function ends). This is obviously not a requirement but it does serve a useful purpose.

```
CD3DInitialize Initialize;

// Create DirectD3D9 object
m_pD3D = Direct3DCreate9( D3D_SDK_VERSION );

// pass it in as the only parameter to our initialization class
Initialize.Enumerate( m_pD3D ) )
```

The Enumerate function does quite a bit of work. It scans the hardware and tests what each adapter on the system is capable of. The class stores a list of adapters. For each adapter in the list, it stores another list with all of the display modes the adapter. It also stores an array of each device type supported (HAL or REF) and for each of those it stores an array of format combinations that work with that device. Each combination contains an adapter format, back buffer format, an array of compatible depth buffer formats that work with this adapter format/back buffer combination. It will determine whether the combination is for windowed or fullscreen mode and store an array of valid presentation intervals that can be used with a given combination. Finally, it stores an array of vertex processing types that can be used with this combination (hardware/software vertex processing or both).

The CD3DInitialize class communicates this data by means of a support class called CD3DSettings (defined in CD3DInitialize.h):

```
class CD3DSettings
{
public:

    struct Settings
    {
        ULONG                   AdapterOrdinal;
        D3DDISPLAYMODE          DisplayMode;
        D3DDEVTYPE              DeviceType;
        D3DFORMAT               BackBufferFormat;
        D3DFORMAT               DepthStencilFormat;
        D3DMULTISAMPLE_TYPE     MultisampleType;
        ULONG                   MultisampleQuality;
        VERTEXPROCESSING_TYPE   VertexProcessingType;
        ULONG                   PresentInterval;
    };

    bool        Windowed;
    Settings    Windowed_Settings;
    Settings    Fullscreen_Settings;

    Settings*   GetSettings(){return(Windowed)? &Windowed_Settings : &Fullscreen_Settings;}
};
```

The class is a container for two Settings structures. Each will hold settings for windowed mode and fullscreen mode. We can pass one of these structures to two member functions of the CD3DInitialization class so that settings can be determined for both windowed and fullscreen modes:

bool CD3DInitialize::FindBestFullscreenMode(CD3DSettings & D3DSettings,
D3DDISPLAYMODE * pMatchMode,
bool bRequireHAL, bool bRequireREF)

This function takes a D3DDISPLAYMODE structure containing the desired display resolution and pixel format. Two optional booleans are provided to indicate whether to enumerate HALs or REF devices. If we do not pass these parameters, the default behavior of the function is to search all devices on all adapters with a preference for a HAL device. When this function returns, the CD3DSetting parameter will have its FullscreenSettings member populated appropriately with available features. We can use this function to find a compatible fullscreen device as follows:

```
    D3DDISPLAYMODE MatchMode;

    // Attempt to find a good default fullscreen set
    MatchMode.Width          = 640;
    MatchMode.Height         = 480;
    MatchMode.Format         = D3DFMT_UNKNOWN;
    MatchMode.RefreshRate    = 0;

    Initialize.FindBestFullscreenMode( m_D3DSettings, &MatchMode );
```

We pass the FindBestFullscreenMode function the CGameApp class CD3Dsettings structure along with a display mode. It searches through its list of adapters/devices and combination arrays and fills the CD3DSettings::FullscreenSettings structure with the closest match it can find to the desired input mode. In this example above, the D3FMT_UNKNOWN format indicates that the application will not be particular about the format used. The same applies to refresh rate. In this case the function will prefer a 640x480 format whose pixel format matches the current adapter display format (desktop format). It will also attempt to use the current desktop refresh rate.

The information stored in the CD3DSetting class can now be used to fill out a D3DPRESENT_PARAMETERS structure and call CreateDevice with a device configuration we can be confident has support.

If our application wants to be able to switch from fullscreen to windowed mode, we should also determine the Windowed settings (for the same CD3DSettings class):

bool CD3DInitialize::FindBestWindowedMode(CD3DSettings & D3DSettings, bool bRequireHAL, bool bRequireREF)

With windowed mode enumeration we will not need to pass the display mode. The code will automatically search for an adapter/device combination that uses the current desktop format, find a compatible frame buffer format, and give preference to HAL devices.

```
// Attempt to find a good default full screen set
MatchMode.Width         = 640;
MatchMode.Height        = 480;
MatchMode.Format        = D3DFMT_UNKNOWN;
MatchMode.RefreshRate   = 0;
Initialize.FindBestFullscreenMode( m_D3DSettings, &MatchMode );

// Attempt to find a good default windowed set
Initialize.FindBestWindowedMode( m_D3DSettings );
```

We can use this function along with its predecessor using the same CD3DSettings object to have both its Windowed_Settings and its Fullscreen_Settings members filled with compatible device creation parameters. This is precisely what we will do in this demo in the CGameApp::CreateDisplay function.

m_D3DSettings.Fullscreen_Settings = settings to create a 640x480 fullscreen device
m_D3DSettings.Windowed_Settings = settings to create a compatible windowed mode device

The above code is used only to provide an initial set of viable default initialization values. The user can then subsequently choose alternatives from a list of video modes that are supported on the current hardware. The results of the user selection will be used to amend the entries in the m_D3DSettings structure to create the requested device.

Note that at this point in the code, no device has been created. The CD3DInitialize::Enumerate function has only built a list of all adapter/device/display mode/depth buffer combinations and stored

them in an array. The FindBestXX functions have returned two of these combinations based on search criteria. The criterion in the example was a 640x480 fullscreen mode in any format and a windowed mode that matches the current display mode.

The next task is to create the device using these settings by calling the CD3DInitialization::CreateDisplay function. The D3DSetting class includes a boolean variable that allows us to specify the desired window mode (windowed vs. fullscreen) so that the appropriate settings are selected during device creation.

The following code snippet searches for an 800x600 32 bit color mode, then retrieves a viable windowed mode, and finally creates a fullscreen device. The windowed setting will be used to reset the device if the user chooses to switch between windowed and fullscreen modes using SHIFT+ENTER.

```
D3DDISPLAYMODE MatchMode;
CD3DInitialize Initialize;

// Create DirectD3D9 object
m_pD3D = Direct3DCreate9( D3D_SDK_VERSION );

// pass it in as the only parameter to our initialization class
Initialize.Enumerate( m_pD3D ) )

// Attempt to find a good default full screen set
MatchMode.Width         = 800;
MatchMode.Height        = 600;
MatchMode.Format        = D3DFMT_A8R8G8B8;
MatchMode.RefreshRate   = 0;
Initialize.FindBestFullscreenMode( m_D3DSettings, &MatchMode );

// Attempt to find a good default windowed set
Initialize.FindBestWindowedMode( m_D3DSettings );

// now state we wish to create a full screen device initially
m_D3Dsettings.Windowed = TRUE

// Create the device and application window
// (parameter list not covered yet, will be discussed next)
LPTSTR          WindowTitle  = _T("Enumeration");
USHORT          Width        = 400;
USHORT          Height       = 400;
Initialize.CreateDisplay(m_D3DSettings, 0, NULL, StaticWndProc, WindowTitle,
                         Width, Height, this );

// Retrieve created items
m_pD3DDevice = Initialize.GetDirect3DDevice( );
m_hWnd       = Initialize.GetHWND( );
```

CD3DInitialize::CreateDisplay

The CreateDisplay function creates the device object and (optionally) the application window. When the function returns successfully, the application will be able to retrieve a pointer to the device

interface that was created and a handle to the window that was created. This is done by calling the GetDirect3DDevice and GetHwnd member functions respectively. These will be stored in the CGameApp class member variables for future application use. At that point, our application could delete (or in our case simply let its scope expire) the CD3DInitialize class if desired.

```
HRESULT CD3DInitialize::CreateDisplay( CD3DSettings& D3DSettings,
                                       ULONG Flags,
                                       HWND hWnd,
                                       WNDPROC pWndProc,
                                       LPCTSTR Title,
                                       ULONG Width,
                                       ULONG Height,
                                       LPVOID lParam )
```

D3DSettings
This is a structure filled with device creation information. The D3DSettings::Windowed boolean will inform the CreateDisplay function whether it should create a fullscreen or windowed device (using its Fullscreen_Settings or Windowed_Settings members respectively).

Flags
The flags passed into this parameter will be passed into the CreateDevice function of the IDirect3D interface after being combined with any other required flags, such as the vertex processing creation flags.

hWnd
If you do not wish the function to create an application window for you automatically, perhaps because you have already created one that has special attributes, you can pass in the HWND of your application window here. This window will be attached to the device as the focus window and the rendering window. If a fullscreen application is being created, this window will be resized to take up the entire display. If NULL is passed instead, the function will use the following four parameters to build a window for you.

pWndProc
If **hWnd** is NULL, then this parameter should be name of the function (often called the WndProc function) that will handle the newly created window's messages. As with all of our demos thus far, this will be the global StaticWndProc function in our application. This function will then dispatch the message to the CGameApp::DisplayWndProc function.

Title
If **hWnd** is NULL, then this string contains the title of the newly created window. The title will be displayed in the caption bar in windowed device mode.

Width
If **hWnd** is NULL, this should contain the desired window width.

Height
If **hWnd** is NULL, this should contain the desired window height.

lParam
If **hWnd** is NULL, this should be a 32 bit value that will be associated with the created window. In chapter 1, we used this per window data to store a pointer to the instance of the CGameApp class to which it belongs. That is also what we will do in this application. We pass in the *this* pointer so that a pointer to the CGameApp class is stored inside the window itself. This is used in the StaticWndProc function to determine which instance of the CGameApp class data should be dispatched to.

If the user wants to switch from a fullscreen device to a windowed device (and vice versa) using SHIFT+ENTER, we can handle this request in our window procedure as follows:

```
case VK_RETURN:

if ( GetKeyState( VK_SHIFT ) & 0xFF00 )
{
    CD3DInitialize Initialize;

     // Toggle full screen / windowed
     m_D3DSettings.Windowed = !m_D3DSettings.Windowed;
     Initialize.ResetDisplay( m_pD3DDevice,
                              m_D3DSettings, m_hWnd );
     SetupRenderStates( );

     // Set menu only in windowed mode
     // (Removed by ResetDisplay automatically in fullscreen)
     if ( m_D3DSettings.Windowed )
     {
         SetMenu( m_hWnd, m_hMenu );
     } // End if Windowed

 } // End if

 break;
```

The CGameApp class has the CD3DSettings stored for both windowed and fullscreen modes. To flip between the two, we toggle the value of the CD3DSetting::Windowed boolean variable and call CInitialize.ResetDisplay. We pass our current device object, the CD3DSetting object with the newly modified boolean, and the HWND of the application window.

Deriving Classes from CD3DInitialize

There are times when our application will require explicit feature support from the 3D hardware. Some examples might be the number of textures the device can blend simultaneously, support for counter-clockwise culling, minimum screen resolutions and so on. The CD3DInitialization class includes a series of overridable virtual functions (which by default all simply return true) to aid in this process:

```
virtual bool ValidateDisplayMode(const D3DDISPLAYMODE& Mode ) { return true; }
virtual bool ValidateDevice(const D3DDEVTYPE& Type, const D3DCAPS9& Caps ) { return true; }
virtual bool ValidateMultiSampleType(const D3DMULTISAMPLE_TYPE& Type){ return true; }
virtual bool ValidatePresentInterval(const ULONG& Interval){ return true; }
virtual bool ValidateDeviceOptions(const D3DFORMAT& BackBufferFormat,bool IsWindowed)
             {return true;}
virtual bool ValidateDepthStencilFormat(const D3DFORMAT& DepthStencilFormat)
             {return true;}
virtual bool ValidateVertexProcessingType(const VERTEXPROCESSING_TYPE& Type)
             {return true;}
```

The general idea behind these ValidateXX calls is that a derived class can override them to reject devices, display modes, depth buffer formats, vertex processing capabilities, etc. The CD3DInitialization will call these functions to determine what choices are valid for your application.

In applications throughout the course, we will override some of these functions to make sure that we reject adapters and devices that do not meet the base requirements. It is unlikely that our applications will reject many devices since most will not require an advanced feature set, but it does set a precedent for future usage.

In CGameApp.h we derive a class from the CD3DInitialization class:

```
class CMyD3DInit : public CD3DInitialize
{
private:
    virtual bool   ValidateDisplayMode   ( const D3DDISPLAYMODE& Mode );
    virtual bool   ValidateDevice        ( const D3DDEVTYPE& Type, const D3DCAPS9& Caps );
    virtual bool   ValidateVertexProcessingType ( const VERTEXPROCESSING_TYPE& Type );
};
```

This derived class overrides three of the validation functions (see CGameApp.cpp).

```
bool CMyD3DInit::ValidateDisplayMode( const D3DDISPLAYMODE &Mode )
{
    // Test display mode
    if ( Mode.Width < 640 || Mode.Height < 480 || Mode.RefreshRate < 60 ) return false;

    // Supported
    return true;
}
```

When we call the CMyD3DInit::Enumerate function, it searches through all adapters on the system and records, in an array, the various video modes that the adapter supports. Before it adds them to the list, it calls the virtual ValidateDisplayMode function and passes in the D3DDISPLAYMODE. Our derived class method checks the width and height of the display and returns true only if the width is not smaller than 640, the height is not smaller than 480, and the refresh rate is not less than 60 Hz. When a mode is encountered where any of the above criteria are not met, we return false back to the Enumerate function of the base class. This instructs it not to add this particular display mode to the list of available display modes. Otherwise, we return true and the display mode is added to the array. The Enumerate function will call this function for every display mode for every adapter on the current

system. This allows us to remove display modes we do not wish to support from consideration. In the above example, a 320x200 video mode would be rejected and thus not be added to the CD3DInitialize database.

ValidateDevice tests the capabilities of the device and rejects those that do not meet the application requirements. This function is called once for every device type found on each adapter during the enumeration function. The input parameters are the device type and the D3DCAPS9 structure containing the capabilities of the device currently being tested by the enumeration function:

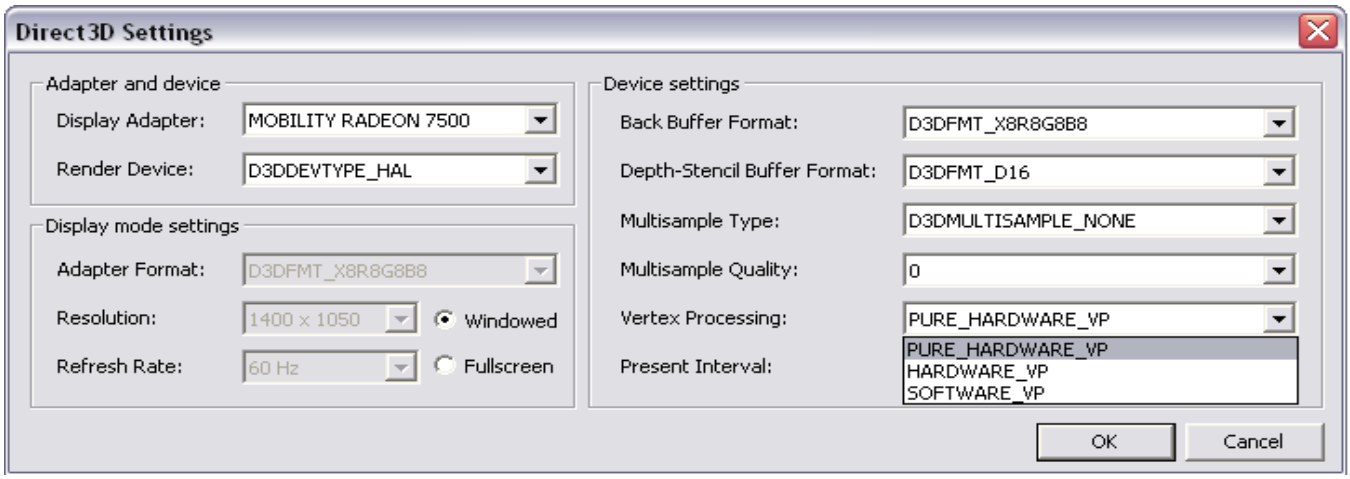
```
bool CMyD3DInit::ValidateDevice( const D3DDEVTYPE &Type, const D3DCAPS9 &Caps )
{
    // Test Capabilities (All device types supported)
    if ( !(Caps.RasterCaps & D3DPRASTERCAPS_DITHER       ) ) return false;
    if ( !(Caps.ShadeCaps & D3DPSHADECAPS_COLORGOURAUDRGB) ) return false;
    if ( !(Caps.PrimitiveMiscCaps & D3DPMISCCAPS_CULLCCW ) ) return false;
    if ( !(Caps.ZCmpCaps & D3DPCMPCAPS_LESSEQUAL         ) ) return false;

    // Supported
    return true;
}
```

Our application checks the D3DCAPS9.RasterCaps member to ensure that dithering is supported. It checks the D3DCAPS9.ShadeCaps member to confirm Gouraud shading. The PrimitiveMiscCaps member is evaluated to make sure that the cull mode we are using (counter clockwise) is supported. Finally, the default Z-Buffer pixel compare of distances is tested. If any of the above tests fail (which is highly unlikely), then the function returns false informing the enumeration function that this device should not be added to the database. The result is that this device will never be used to search for compatible video modes during the CD3DInitialize::FindBestXX functions.

The last function we override in our demo is the ValidateVertexProcessingType. It allows us to reject any devices that do not support the required vertex processing support (eg. hardware transformation and lighting). It can also be used to remove unnecessary processing types. For example, our application does not need a device that supports mixed vertex processing.

```
bool CMyD3DInit::ValidateVertexProcessingType( const VERTEXPROCESSING_TYPE &Type )
{
    // Test Type ( We don't need mixed  )
    if ( Type == MIXED_VP ) return false;

    // Supported
    return true;
}
```

When you run the Lab Project 2.2 application a dialog box will appear. You will be able to select the various device parameters you desire. Notice that in the Vertex Processing combo box, mixed vertex processing is not an option (in keeping with the function above):



The CD3DInitialization Class

The CD3DInitialization class and all of its support classes are defined in CD3DInitialization.cpp:

```
class CD3DInitialize
{
public:
 // Member function have been snipped from here
private
 // Member variables
    LPDIRECT3D9             m_pD3D;                 // Primary Direct3D Object.
    LPDIRECT3DDEVICE9       m_pD3DDevice;           // Created Direct3D Device.
    HWND                    m_hWnd;                 // Created window handle
    VectorAdapter           m_vpAdapters;           // Enumerated Adapters
};
```

The first member is a pointer to the IDirect3D9 interface passed into the CD3DInitialize function. The second holds the IDirect3DDevice9 interface to the device object that will ultimately be created at the end of the enumeration process (when the user calls CD3DInitialize::CreateDisplay). The third HWND parameter will eventually be filled with the handle to the automatically created window created by this class in the CreateDisplay member function. All three of these member variables will initially be set to NULL. The fourth parameter stores an array of CD3DEnumAdapter class pointers. This class is a support class which has members to identify each adapter on the current system. VectorAdapter is a typedef for an STL vector of pointers of type CD3DEnumAdpater:

```
typedef std::vector<CD3DEnumAdapter*>  VectorAdapter;
```

Please refer to the appendices for a brief refresher if you are unfamiliar with the STL vector type.

CD3DEnumAdpater is a support class that maintains adapter information. Usually, there will only be one adapter on most computer systems and after enumeration has taken place, the m_vpAdapters vector will only hold a single pointer to a CD3DEnumAdapter class. When there is more than one

adapter on the current system, there will be one CD3DEnumAdpater class generated for each adapter on the system.

```
class CD3DEnumAdapter
{
public:
    ~CD3DEnumAdapter();

    ULONG                     Ordinal;
    D3DADAPTER_IDENTIFIER9    Identifier;
    VectorDisplayMode         Modes;
    VectorDevice              Devices;
};
```

Each CD3DEnumAdpater instance holds an adapter ordinal and the adapter identifier. We can retrieve an adapter identifier using the IDirect3D9::GetAdapterIdentifier function. D3DADAPTER_INDENTIFIER9 holds information about the hardware such as the name of the card, the driver, and the manufacturer:

```
typedef struct _D3DADAPTER_IDENTIFIER9
{
    char Driver[MAX_DEVICE_IDENTIFIER_STRING];
    char Description[MAX_DEVICE_IDENTIFIER_STRING];
    char DeviceName[32];
    LARGE_INTEGER DriverVersion;
    DWORD DriverVersionLowPart;
    DWORD DriverVersionHighPart;
    DWORD VendorId;
    DWORD DeviceId;
    DWORD SubSysId;
    DWORD Revision;
    GUID DeviceIdentifier;
    DWORD WHQLLevel;
} D3DADAPTER_IDENTIFIER9;
```

Our application is interested only in the ***Description*** field. This is a string that tells us the name of the adapter (ex. ATI Radeon 7500™, nVidia geForce 3™, etc.). The user will be able to see that the application is using the desired adapter.

Each CD3DEnumAdapter also manages two vectors. The first will be filled with all of the display modes that the adapter supports and is defined as:

```
typedef std::vector<D3DDISPLAYMODE> VectorDisplayMode;
```

This vector holds an array of D3DDISPLAYMODE structures. Each D3DDISPLAYMODE contains a width, height, pixel format, and refresh rate.

The second vector is defined as:

```
typedef std::vector<CD3DEnumDevice*> VectorDevice;
```

VectorDevice contains CD3DEnumDevice class pointers. This class will be used to hold device information. As mentioned previously, several device types might be supported. For example, for most adapters you will usually be able to create a REF device and a HAL device type. Each device type that is available for an adapter will be stored in the CD3DEnumAdapter::VectorDevice array.

You should begin to see a hierarchy forming here. At the root there is a list of adapters (CD3DEnumAdpater classes). For each adapter there is a list of display modes and a list of devices available on that adapter (CD3DenumDevice classes).

The CD3DEnumDevice class is also defined in CD3DInitialize.h:

```
class CD3DEnumDevice
{
public:
    ~CD3DEnumDevice();

    D3DDEVTYPE                DeviceType;
    D3DCAPS9                  Caps;
    VectorDeviceOptions       Options;
};
```

For each device, this class is used to store the device type, device capabilities, and a vector of options available on that device. In this context, a device option is a compatible combination of display modes, frame buffer formats, pixel formats, refresh rates, depth buffer formats, presentation intervals, vertex processing capabilities and multi-sampling modes available on that device.

VectorDeviceOptions is type defined as:

```
typedef std::vector<CD3DEnumDeviceOptions*> VectorDeviceOptions;
```

Each CD3DEnumDevice class maintains an array of CD3DEnumDeviceOptions class pointers for each configuration combination that works with this device. This class is shown below:

```
class CD3DEnumDeviceOptions
{
public:
    ~CD3DEnumDeviceOptions();

    ULONG                   AdapterOrdinal;
    D3DDEVTYPE              DeviceType;
    D3DCAPS9                Caps;
    D3DFORMAT               AdapterFormat;
    D3DFORMAT               BackBufferFormat;
    bool                    Windowed;
    VectorMSType            MultiSampleTypes;
    VectorULONG             MultiSampleQuality;
    VectorFormat            DepthFormats;
    VectorVPType            VertexProcessingTypes;
    VectorULONG             PresentIntervals;
};
```

The first three members of this structure contain duplicated device and adapter information from further up the hierarchy. It is convenient to store the adapter and device at this level because this is the structure our FindBestWindowedMode and FindBestFullscreenMode functions will be working with when searching for a suitable device and mode. In the end, these functions are searching for a compatible DeviceOptions structure with which to create the device. When we enumerate our adapters, each device on that adapter will have a structure allocated for every unique combination of settings that can be used with that device.

Each device option contains an adapter format and back buffer format that the device is compatible with. The Windowed boolean identifies the option as either a windowed or fullscreen mode setting combination. This is ultimately what a combination is. It is a front buffer/back buffer windowed or fullscreen configuration that works with a given device. For each front buffer/back buffer combination there will be a new option added to the array for the device.

The remaining members are arrays of other settings that work with the combination. There is an unique array for depth buffer formats, another for presentation intervals, one for vertex processing settings, and an array of multi-sampling capabilities used for anti-aliasing. These vectors are type defined to hold core DirectX structures as seen below:

```
typedef std::vector<D3DMULTISAMPLE_TYPE>    VectorMSType;
typedef std::vector<D3DFORMAT>              VectorFormat;
typedef std::vector<ULONG>                  VectorULONG;
typedef std::vector<VERTEXPROCESSING_TYPE>  VectorVPType;
```

This entire hierarchy will be filled with information by the CD3DInitialization class using the Enumerate function. The function will enter a loop where it looks for available adapters. For each adapter found it adds a CD3DEnumAdpater class to the CD3DInitialize adapter array. It enumerates all of the available display modes and stores them in a mode array inside the CD3DEnumAdapater class.

For each adapter, it loops through all of the device types it is capable of supporting. For each device it allocates a CD3DEnumDevice class and adds it to the CD3DEnumAdapter device array. Another loop finds every supported front buffer/back buffer format combination that can be used with the device and stores that information in CD3DEnumDevice::CD3DEnumDeviceOptions. This structure is filled with information such as viable depth buffer formats, vertex processing types, and so on.

Once all of this information has been enumerated, our application can call CD3DInitialize::FindBestFullscreenMode. That function can now loop through each adapter, each device for that adapter, and eventually each device option set stored for that device until it finds one that best suits its needs.

Fig 2.3 depicts how the classes just discussed are stored in a hierarchy that can be navigated to find the best results. Keep in mind that before a single display mode, device, or device option set is added to the hierarchy, the Validation functions are called. These functions provide the application an opportunity to accept or reject a given set of capabilities.

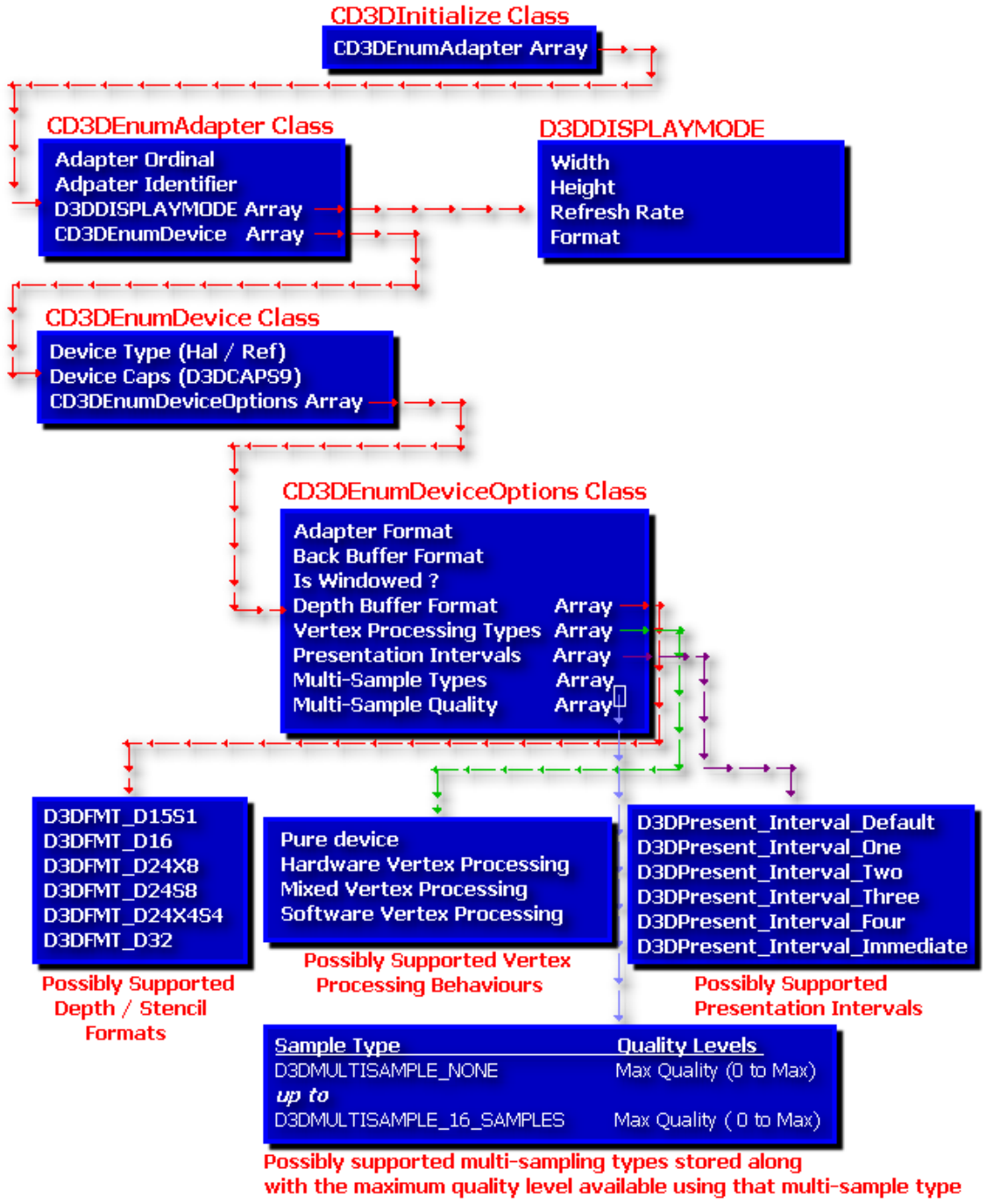

CD3DInitialize Class
CD3DEnumAdapter Array
CD3DEnumAdapter Class
Adapter Ordinal
Adpater Identifier
D3DDISPLAYMODE Array
CD3DEnumDevice Array
D3DDISPLAYMODE
Width
Height
Refresh Rate
Format
CD3DEnumDevice Class
Device Type (Hal / Ref)
Device Caps (D3DCAPS9)
CD3DEnumDeviceOptions Array
CD3DEnumDeviceOptions Class
Adapter Format
Back Buffer Format
Is Windowed ?
Depth Buffer Format Array
Vertex Processing Types Array
Presentation Intervals Array
Multi-Sample Types Array
Multi-Sample Quality Array
D3DFMT_D15S1
D3DFMT_D16
D3DFMT_D24X8
D3DFMT_D24S8
D3DFMT_D24X4S4
D3DFMT_D32
Possibly Supported Depth / Stencil Formats
Pure device
Hardware Vertex Processing
Mixed Vertex Processing
Software Vertex Processing
Possibly Supported Vertex Processing Behaviours
D3DPresent_Interval_Default
D3DPresent_Interval_One
D3DPresent_Interval_Two
D3DPresent_Interval_Three
D3DPresent_Interval_Four
D3DPresent_Interval_Immediate
Possibly Supported Presentation Intervals
Sample Type
Quality Levels
D3DMULTISAMPLE_NONE
Max Quality (0 to Max)
up to
D3DMULTISAMPLE_16_SAMPLES
Max Quality (0 to Max)
Possibly supported multi-sampling types stored along with the maximum quality level available using that multi-sample type

Figure 2.3

CD3DInitialize::Enumerate

The enumerate function is the first CD3DInitialize function explicitly called by our application. It initiates the construction of the adapter/device options database:

```
HRESULT CD3DInitialize::Enumerate( LPDIRECT3D9 pD3D )
{
    HRESULT hRet;

    // Store the D3D Object
    m_pD3D = pD3D;
    if ( !m_pD3D ) return E_FAIL;

    // We have made copy of pointer do increase reference count
    m_pD3D->AddRef();

    // Enumerate the adapters
    if ( FAILED( hRet = EnumerateAdapters() ) ) return hRet;

    // Success!
    return S_OK;
}
```

The function stores the input IDirect3D9 pointer for later use and then calls the EnumerateAdapters member function. This function is responsible for filling up the CD3DEnumAdpater array:

CD3DInitialize::EnumerateAdapters

```
HRESULT CD3DInitialize::EnumerateAdapters()
{
    HRESULT hRet;

    // Store the number of available adapters
    ULONG nAdapterCount = m_pD3D->GetAdapterCount();

    // Loop through each adapter
    for ( ULONG i = 0; i < nAdapterCount; i++ )
    {
        CD3DEnumAdapter * pAdapter = new CD3DEnumAdapter;
        if ( !pAdapter ) return E_OUTOFMEMORY;

        // Store adapter ordinal
        pAdapter->Ordinal = i;

        // Retrieve adapter identifier
        m_pD3D->GetAdapterIdentifier( i, 0, &pAdapter->Identifier );

        // Enumerate all display modes for this adapter
        if ( FAILED( hRet = EnumerateDisplayModes( pAdapter ) ) ||
             FAILED( hRet = EnumerateDevices( pAdapter ) ))
        {
            delete pAdapter;
            if ( hRet == E_ABORT ) continue; else return hRet;
```

```
        } // End if Failed Code

        // Add this adapter the list
        try { m_vpAdapters.push_back( pAdapter ); } catch ( ... )
        {
            delete pAdapter;
            return E_OUTOFMEMORY;

        } // End Try / Catch Block

    } // Next Adapter

    // Success!
    return S_OK;
}
```

To begin, we query the Direct3D9 object for the number of graphics adapters installed on the system. For each adapter, we allocate a CD3DEnumAdpater class to be filled with the adapter information. We record the adapter ordinal and call the IDirect3D::GetAdapterIdentifier method to retrieve the adapter details (name, description, driver version, etc.). The CD3DEnumAdapter class has two arrays to be filled. The first contains all of the supported display modes and the second contains a list of supported device types (encapsulated by the CD3DEnumDevice class). To fill these arrays we call two more methods of this class: EnumerateDisplayModes and EnumerateDevices. When these functions return, the CD3DEnumAdapter class contains all necessary information. Finally, we add the adapter to the vector. At this point, the enumeration process is complete and the adapter/device database has been built.

Most of the work happens in the two helper functions called from the above code. CD3DInitialize::EnumerateDisplayModes is responsible for compiling an array of display modes supported by the device. The following function loops through all supported Direct3D pixel formats. There are a limited number of D3DFMT types that can be used for the physical adapter mode:

```
const ULONG          ValidAdapterFormatCount = 3;
const D3DFORMAT      ValidAdapterFormats[3]  = { D3DFMT_X8R8G8B8, D3DFMT_X1R5G5B5,
                                                 D3DFMT_R5G6B5 };
```

At least one of these adapter formats will be supported by all Direct3D compatible cards (possibly all three). Our function needs to loop through all three formats in the array. It needs to enumerate all of the display modes (width, height, and refresh rates) that the device can support in that format.

CD3DInitialize::EnumerateDisplayModes

```
HRESULT CD3DInitialize::EnumerateDisplayModes( CD3DEnumAdapter * pAdapter )
{
    HRESULT         hRet;
    ULONG           i, j;
    D3DDISPLAYMODE  Mode;
```

```
    // Loop through each valid 'Adapter' format.
    for ( i = 0; i < ValidAdapterFormatCount; i++ )
    {
        // Retrieve the number of valid modes for this format
        ULONG nModeCount = m_pD3D->GetAdapterModeCount( pAdapter->Ordinal,
                                                        ValidAdapterFormats[i] );
        if ( nModeCount == 0 ) continue;

        // Loop through each display mode for this format
        for ( j = 0; j < nModeCount; j++ )
        {
            // Retrieve the display mode
            hRet = m_pD3D->EnumAdapterModes( pAdapter->Ordinal, ValidAdapterFormats[i],
                                             j, &Mode );
            if ( FAILED( hRet ) ) return hRet;

            // Is supported by user ?
            if ( !ValidateDisplayMode( Mode ) ) continue;

            // Add this mode to the adapter
            try { pAdapter->Modes.push_back( Mode ); } catch( ... )
            {
                return E_OUTOFMEMORY;
            } // End Try / Catch block

        } // Next Adapter Mode

    } // Next Adapter Format

    // Success?
    return (pAdapter->Modes.size() == 0) ? E_ABORT : S_OK;
}
```

We test each adapter format in the array, and call the IDirect3D9::GetAdapterModeCount function to retrieve the number of display modes that the adapter supports in that pixel format. If the count is zero, then the device does not support the format and we move on to the next iteration of the loop. If the result is non-zero, then this is the number of different display mode combinations the adapter can support with that pixel format. Next, we loop through each of these modes and call the IDirect3DDevice9::EnumAdpaterModes function passing the format and the mode number to retrieve the display mode.

Before adding the returned display mode to the CD3DEnumAdapter display mode array, the virtual function ValidateDisplayMode is called. If the derived function returns true, then the display mode is added to the array. Once done, we retrieve the size of the vector. A size of zero means that all display modes were rejected. In this case we would return E_ABORT to tell the EnumerateAdapters function to remove this adapter from further consideration.

When the overall process is complete, program flow returns to the CD3DInitialize::EnumerateAdapters function. There now exists an array of valid display modes and CD3DInitialize::EnumerateDevices is called to fill the array of device types for CD3DEnumAdapter.

CD3DInitialize::EnumerateDevices

This function will populate the CD3DEnumAdpater::CD3DEnumDevices array by traversing all of the DirectX supported device types to test whether they are valid for this adapter. At the top of the .cpp file, we define an array holding the three possible device types:

```
const ULONG            DeviceTypeCount = 3;
const D3DDEVTYPE       DeviceTypes[3] = { D3DDEVTYPE_HAL, D3DDEVTYPE_SW, D3DDEVTYPE_REF };
```

Generally only a HAL or REF device will be available since third party software devices (D3DDEVTYPE_SW) do not ship with DirectX9 and are not widely used.

The function loops through each device type supported by DirectX and calls IDirect3D9::GetDeviceCaps to retrieve the device type capabilities on the current adapter. If the device type is not available on the adapter, the function will fail and the loop continues. If the call succeeds, then the device type is supported by the adapter and the D3DCAPS9 structure will be filled with all the device capabilities info. Finally, the function calls the virtual function ValidateDevice to allow the application to accept or reject the passed device type.

```
HRESULT CD3DInitialize::EnumerateDevices( CD3DEnumAdapter * pAdapter )
{
    ULONG    i;
    HRESULT  hRet;
    D3DCAPS9 Caps;

    // Loop through each device type (HAL, SW, REF)
    for ( i = 0; i < DeviceTypeCount; i++ )
    {
        // Retrieve device caps (on failure, device not generally available)
        if ( FAILED( m_pD3D->GetDeviceCaps( pAdapter->Ordinal, DeviceTypes[i], &Caps ) ) )
            continue;

        // Supported by user ?
        if ( !ValidateDevice( DeviceTypes[ i ], Caps ) ) continue;

        // Allocate a new device
        CD3DEnumDevice * pDevice = new CD3DEnumDevice;
        if ( !pDevice ) return E_OUTOFMEMORY;

        // Store device information
        pDevice->DeviceType = DeviceTypes[i];
        pDevice->Caps       = Caps;

        // Retrieve various init options for this device
        if ( FAILED( hRet = EnumerateDeviceOptions( pDevice, pAdapter ) ) )
        {
            delete pDevice;
            if ( hRet == E_ABORT ) continue; else return hRet;

        } // End if failed to enumerate

        // Add it to our adapter list
        try { pAdapter->Devices.push_back( pDevice ); } catch ( ... )
```

```
        {
            delete pDevice;
            return E_OUTOFMEMORY;

        } // End Try / Catch Block

    } // Next Device Type

    // Success?
    return (pAdapter->Devices.size() == 0) ? E_ABORT : S_OK;

}
```

If ValidateDevice returns true, a new CD3DEnumDevice is allocated and will be added to the CD3DEnumAdapter device array. The object is filled with the device type and capabilities previously retrieved into the class member variables. Before adding it to the device array, we call the EnumerateDeviceOptions function to compile an array of device options (stored in CD3DEnumDevice::Options). When EnumerateDeviceOptions returns, there will be an option set for every compatible adapter/back buffer format the device is capable of (both windowed and fullscreen mode).

CD3DInitialize::EnumerateDeviceOptions

CD3DEnumDeviceOptions stores five vectors that are used to store various options that can be used with each adapter/back buffer format combination. We declare an array of all possible back buffer formats supported by DirectX9 at the top of the .cpp file to make it easier to loop and test them all:

```
const ULONG       BackBufferFormatCount   = 11;
const D3DFORMAT   BackBufferFormats[11]   = { D3DFMT_R8G8B8,      D3DFMT_A8R8G8B8,
                                              D3DFMT_X8R8G8B8,  D3DFMT_R5G6B5,
                                              D3DFMT_A1R5G5B5,  D3DFMT_X1R5G5B5,
                                              D3DFMT_R3G3B2,      D3DFMT_A8R3G3B2,
                                              D3DFMT_X4R4G4B4,   D3DFMT_A4R4G4B4,
                                              D3DFMT_A2B10G10R10 };

HRESULT CD3DInitialize::EnumerateDeviceOptions(CD3DEnumDevice *pDevice,
                                               CD3DEnumAdapter *pAdapter)
{
    HRESULT     hRet;
    ULONG       i, j, k;
    bool        Windowed;
    D3DFORMAT   AdapterFormats[ ValidAdapterFormatCount ];
    ULONG       AdapterFormatCount = 0;
    D3DFORMAT   AdapterFormat, BackBufferFormat;

    // Build a list of all the formats used by the adapter
    for ( i = 0; i < pAdapter->Modes.size(); i++ )
    {
        // Already added to the list ?
        for ( j = 0; j < AdapterFormatCount; j++ )
            if ( pAdapter->Modes[i].Format == AdapterFormats[j] ) break;
```

```
        // Add it to the list if not existing.
        if ( j == AdapterFormatCount )
            AdapterFormats[ AdapterFormatCount++ ] = pAdapter->Modes[i].Format;

    } // Next Adapter Mode
```

This function starts by building a local list of possible adapter formats supported by the current CD3DEnumAdapter object. These formats are stored in the local array AdapterFormats. Recall that the CD3DEnumAdapter object stored all of the display modes in an array. So it is a case of testing each format and adding it to the local array (avoiding duplicates).

At this point the function has an array of adapter format supported by the device. It will now iterate over the list and test each adapter format against every possible back buffer format in our const array (see inner loop). After it has determined an adapter format and a back buffer format, it will check whether this combination is supported by the device in both windowed and fullscreen modes. The combination is evaluated using the IDirect3D9::CheckDeviceType function. This function will only succeed when the adapter format and back buffer format can be used together on the current device. If the function does succeed, a new CD3DEnumDeviceOptions object is created and its member variables will be populated.

```
    // Loop through each adapter format available
    for ( i = 0; i < AdapterFormatCount; i++ )
    {
        // Store Adapter Format
        AdapterFormat = AdapterFormats[i];

        // Loop through all valid back buffer formats
        for ( j = 0; j < BackBufferFormatCount; j++ )
        {
            // Store Back Buffer Format
            BackBufferFormat = BackBufferFormats[j];

            // Test Windowed / Fullscreen Modes
            for ( k = 0; k < 2; k++ )
            {
                // Select windowed / fullscreen
                if ( k == 0 ) Windowed = false; else Windowed = true;

                // Skip if this is not a valid device type
                if ( FAILED( m_pD3D->CheckDeviceType(pAdapter->Ordinal,
                                                     pDevice->DeviceType,
                                                     AdapterFormat,
                                                     BackBufferFormat, Windowed)))

                      continue;

                // Allocate a new device options set
                CD3DEnumDeviceOptions * pDeviceOptions = new CD3DEnumDeviceOptions;
                if (!pDeviceOptions) return E_OUTOFMEMORY;

                // Store device option details
                pDeviceOptions->AdapterOrdinal     = pAdapter->Ordinal;
                pDeviceOptions->DeviceType         = pDevice->DeviceType;
                pDeviceOptions->AdapterFormat      = AdapterFormat;
```

```
                pDeviceOptions->BackBufferFormat   = BackBufferFormat;
                pDeviceOptions->Caps               = pDevice->Caps;
                pDeviceOptions->Windowed           = Windowed;
```

Although CD3DEnumDeviceOptions is a child of CD3DEnumDevice (which is itself a child of CD3DEnumAdpater), we still store a copy of the adapter ordinal and the device type at this level in the hierarchy. This makes the class more user-friendly since it removes the requirement to maintain links back up the hierarchy to its parents.

ValidateDeviceOptions is called next and affords the application an opportunity to reject this particular option from being added to the database.

```
                // Is this option set supported by the user ?
                if ( !ValidateDeviceOptions( BackBufferFormat, Windowed ) )
                {
                    delete pDeviceOptions;
                    continue;
                } // End if user-unsupported
```

The derived function is passed the back buffer format and a windowed mode boolean. Perhaps your application might derive a function that rejects all 16 bit windowed formats or perhaps even all windowed modes if it only wanted to allow fullscreen gaming. If the ValidateDeviceOptions function returns false, then this particular set of options is not added to the CD3DEnumDevice options array.

As the function reaches the bottom of the adapter database hierarchy, it calls four more short enumeration functions. Recall that each option set maintains a number of arrays: compatible depth/stencil formats, multi-sampling capabilities, vertex processing behavior flags, and presentation intervals. Each of the following enumeration functions are used to populate these arrays:

```
                // Enumerate the various options components
                if ( FAILED( hRet = EnumerateDepthStencilFormats  ( pDeviceOptions ) ) ||
                     FAILED( hRet = EnumerateMultiSampleTypes     ( pDeviceOptions ) ) ||
                     FAILED( hRet = EnumerateVertexProcessingTypes( pDeviceOptions ) ) ||
                     FAILED( hRet = EnumeratePresentIntervals     ( pDeviceOptions ) ) )
                {
                    // Release our invalid options
                    delete pDeviceOptions;

                    // If returned anything other than abort, this is fatal
                    if ( hRet == E_ABORT ) continue; else return hRet;

                } // End if any enumeration failed

                // Add this to our device
                try { pDevice->Options.push_back( pDeviceOptions ); } catch ( ... )
                {
                    delete pDeviceOptions;
                    return E_OUTOFMEMORY;

                } // End Try / Catch Block

            } // Next Windowed State
```

```
        } // Next BackBuffer Format

    } // Next Adapter Format

    // Success?
    return (pDevice->Options.size() == 0) ? E_ABORT : S_OK;
}
```

If all four functions return true, then the four arrays will store a complete set of options that work with this device and the option set will be added to the device option array. The Enumeration process is now officially finished and control is handed back to the calling application. For completeness, we will examine the four enumeration functions used to fill out the option set arrays.

CD3DInitialize::EnumerateDepthStencilFormats

EnumerateDepthStencil format uses a global array of all the possible Direct3D depth/stencil formats to loop through to test with the current option set. The array is declared at the top of CD3DInitialize.cpp:

```
const ULONG        DepthStencilFormatCount = 6;
const D3DFORMAT    DepthStencilFormats[6]  = { D3DFMT_D16, D3DFMT_D15S1, D3DFMT_D24X8,
                                               D3DFMT_D24S8, D3DFMT_D24X4S4, D3DFMT_D32 };

HRESULT CD3DInitialize::EnumerateDepthStencilFormats(CD3DEnumDeviceOptions *pDeviceOptions)
{
    ULONG i;

    try
    {
        // Loop through each depth stencil format
        for ( i = 0; i < DepthStencilFormatCount; i++ )
        {
            // Test to see if this is a valid depth surface format
            if ( SUCCEEDED( m_pD3D->CheckDeviceFormat(pDeviceOptions->AdapterOrdinal,
                                                      pDeviceOptions->DeviceType,
                                                      pDeviceOptions->AdapterFormat,
                                                      D3DUSAGE_DEPTHSTENCIL,
                                                      D3DRTYPE_SURFACE,
                                                      DepthStencilFormats[ i ])))
            {
                // Test to see if this is a valid depth / stencil format for this mode
                if(SUCCEEDED(m_pD3D->CheckDepthStencilMatch(pDeviceOptions->AdapterOrdinal,
                                                            pDeviceOptions->DeviceType,
                                                            pDeviceOptions->AdapterFormat,
                                                            pDeviceOptions->BackBufferFormat,
                                                            DepthStencilFormats[ i ] ) ) )
                {
                    // Is this supported by the user ?
                    if ( ValidateDepthStencilFormat( DepthStencilFormats[ i ] ) )
                    {
                        // Add this as a valid depthstencil format
                        pDeviceOptions->DepthFormats.push_back( DepthStencilFormats[ i ] );
```

```
                    } // End if User-Supported

                } // End if valid for this mode

            } // End if valid DepthStencil format

        } // Next DepthStencil Format

    } // End Try Block

    catch ( ... ) { return E_OUTOFMEMORY; }

    // Success ?
    return ( pDeviceOptions->DepthFormats.size() == 0 ) ? E_ABORT : S_OK;
}
```

If CheckDeviceFormat succeeds (the device supports this depth format) and CheckDepthStencilMatch also succeeds (the depth buffer can be used with this device option set), then a ValidateXX derived call is made before adding the set to the passed CD3DEnumDeviceOptions object depth/stencil format array. The derived function allows the application to reject depth/stencil formats it does not wish to be considered.

CD3DInitialize::EnumerateMultiSampleTypes

Each pixel on our screen represents a color sample taken from the 3D scene. Imagine a ray cast out from each pixel on the monitor directly into the scene. Where the ray intersects a particular object in the scene, the color of the object at that intersection point will be used as the pixel color at that screen location. We call this process **sampling**. As the number of available pixels increases (at higher screen resolutions), the number of samples that will be taken increases. Higher sampling frequencies result in sharper images and more accurate representations of the scene are the result. When fewer samples are taken, we simply have less information to work with to represent the scene and there is less detail that can be displayed. Low sampling frequencies result in image artifacts. One of the most common is a jagged diagonal line (‘the jaggies’). This artifact is recognizable by its staircase like appearance.

The issue is essentially one of trying to represent (or *alias*) something that is infinite (sampling the world at every possible location would result in a precise representation of the world) with something that is finite (the number of available pixels that can be filled with color).

Anti-aliasing algorithms such as multi-sampling are used to combat artifacts that occur at low sample frequencies. When a device supports multi-sampling, it means that it has the ability to rescan the front buffer again (possibly multiple times) and detect where jagged lines occur. It will then smooth them out by blending together colors from neighboring pixels.

Consider the simple scenario of a white diagonal line rendered on a pure black background. In a low resolution mode the line would appear to step up across the screen like a set of stairs. Rather than looking like a diagonal line, it actually looks like a series of horizontal and vertical lines placed in such a way that it represents a best fit approximation of the diagonal line. When multi-sampling is used, the

hardware can detect where the jaggies occur and insert some in-between colored pixels (gray) to help blend the jagged edge into the back ground color. This simple solution can vastly improve the visual quality of the scene.

Support for multi-sampling varies across video hardware. Some cards perform no multi-sampling at all while newer cards often support one or more sampling passes. The tradeoff is additional processing and some loss in application performance. Anti-aliasing is a relatively demanding task even for the latest hardware. Although DirectX Graphics supports up to 16 blending passes, most graphics adapters currently support one or two passes at most.

The EnumerateMultiSampleTypes function will test all DirectX Graphics multi-sample types against the current device option. We define an array at the top of CD3DInitialize.cpp containing all the multi-sampling types supported by DirectX Graphics. If the type is supported by the device option set, it is added to the CD3DEnumDeviceOption multi-sample array:

```
const ULONG                 MultiSampleTypeCount    = 17;
const D3DMULTISAMPLE_TYPE   MultiSampleTypes[17]    = {
                              D3DMULTISAMPLE_NONE      ,
                              D3DMULTISAMPLE_NONMASKABLE,
                              D3DMULTISAMPLE_2_SAMPLES , D3DMULTISAMPLE_3_SAMPLES,
                              D3DMULTISAMPLE_4_SAMPLES , D3DMULTISAMPLE_5_SAMPLES,
                              D3DMULTISAMPLE_6_SAMPLES , D3DMULTISAMPLE_7_SAMPLES,
                              D3DMULTISAMPLE_8_SAMPLES , D3DMULTISAMPLE_9_SAMPLES,
                              D3DMULTISAMPLE_10_SAMPLES, D3DMULTISAMPLE_11_SAMPLES,
                              D3DMULTISAMPLE_12_SAMPLES, D3DMULTISAMPLE_13_SAMPLES,
                              D3DMULTISAMPLE_14_SAMPLES, D3DMULTISAMPLE_15_SAMPLES,
                              D3DMULTISAMPLE_16_SAMPLES };
```

CD3DEnumDeviceOption also includes a linked array that holds a maximum quality setting for each multi sample type. For example, we may have a device option set that allows us to use two samples (D3DMULTISAMPLE_2_SAMPLE) but that supports three quality levels. If we use D3DMULTISAMPLE_2_SAMPLE, we also have a choice of setting the quality to 0, 1, 2 or 3. The higher number provides better visual quality at the cost of performance.

This next function checks available sampling capabilities. If one or more of the multi-sampling types is supported by the hardware, its maximum quality is also returned and stored in a separate array:

```
HRESULT CD3DInitialize::EnumerateMultiSampleTypes( CD3DEnumDeviceOptions * pDeviceOptions )
{
    ULONG i, Quality;
    try
    {
        // Loop through each multi-sample type
        for ( i = 0; i < MultiSampleTypeCount; i++ ){
            // Check if this multi-sample type is supported
            if(SUCCEEDED(m_pD3D->CheckDeviceMultiSampleType(pDeviceOptions->AdapterOrdinal,
                                                            pDeviceOptions->DeviceType,
                                                            pDeviceOptions->BackBufferFormat,
                                                            pDeviceOptions->Windowed,
                                                            MultiSampleTypes[i], &Quality)))
            {
```

```
                // Is this supported by the user ?
                if ( ValidateMultiSampleType( MultiSampleTypes[ i ] ) )
                {
                    // Supported, add these to our list
                    pDeviceOptions->MultiSampleTypes.push_back( MultiSampleTypes[i] );
                    pDeviceOptions->MultiSampleQuality.push_back( Quality );

                } // End if User-Supported

            } // End if valid for this mode

        } // Next Sample Type

    } // End try Block

    catch ( ... ) { return E_OUTOFMEMORY; }

    // Success ?
    return ( pDeviceOptions->MultiSampleTypes.size() == 0 ) ? E_ABORT : S_OK;
}
```

For every supported multi-sample type, we call the IDirect3D9::CheckDeviceMultiSample function passing in the adapter, device, back buffer format and window mode. If a type is supported and the function is successful, then the DWORD variable passed into the function as the last parameter will hold the maximum quality for that multi sample type for this device option set. Finally, we call the ValidateMultiSampleType function. If it returns true, we add the multi-sample type to the device option multi-sample array and add the maximum quality to the matching array. When this function returns, the current CD3DEnumDeviceOptions object will have its multi-sample type array and its multi-sample quality array filled with all supported multi-sampling modes and their maximum qualities.

CD3DInitialize::EnumerateVertexProcessingTypes

This function is called from EnumerateDeviceOptions and is used to fill an array belonging to the CD3DEnumDeviceOptions class with the vertex processing behavior that works with the current device options set. It checks the D3DCAP9 structure of the current device to see which processing modes are available and adds them the array. The following enumerated type is declared in CD3DInitialize.h and is used in this function:

```
enum VERTEXPROCESSING_TYPE
{
    SOFTWARE_VP               = 1,        // Software Vertex Processing
    MIXED_VP                  = 2,        // Mixed Vertex Processing
    HARDWARE_VP               = 3,        // Hardware Vertex Processing
    PURE_HARDWARE_VP          = 4         // Pure Hardware Vertex Processing
};

HRESULT CD3DInitialize::EnumerateVertexProcessingTypes(
                                                    CD3DEnumDeviceOptions* pDeviceOptions)
{
    try{
```

```
        // If the device supports Hardware T&L
        if ( pDeviceOptions->Caps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT )
        {
            // If the device can be created as 'Pure'
            if ( pDeviceOptions->Caps.DevCaps & D3DDEVCAPS_PUREDEVICE )
            {
                // Supports Pure hardware device ?
                if ( ValidateVertexProcessingType( PURE_HARDWARE_VP ) )
                    pDeviceOptions->VertexProcessingTypes.push_back( PURE_HARDWARE_VP );

            } // End if

            // Supports hardware T&L and Mixed by definition ?
            if ( ValidateVertexProcessingType( HARDWARE_VP ) )
                pDeviceOptions->VertexProcessingTypes.push_back( HARDWARE_VP );

            if ( ValidateVertexProcessingType( MIXED_VP ) )
                pDeviceOptions->VertexProcessingTypes.push_back( MIXED_VP );

        } // End if HW T&L

        // Always supports software
        if ( ValidateVertexProcessingType( SOFTWARE_VP ) )
            pDeviceOptions->VertexProcessingTypes.push_back( SOFTWARE_VP );

    } // End try Block

    catch ( ... ) { return E_OUTOFMEMORY; }

    // Success ?
    return ( pDeviceOptions->VertexProcessingTypes.size() == 0 ) ? E_ABORT : S_OK;
}
```

The function calls the virtual function ValidateVertexProcessingType which the derived class uses to accept or reject vertex processing types. For example, in this demo, the derived class returns false for MIXED_VP processing behavior so that mixed vertex mode options will not be added to the internal enumeration database.

CD3DInitialize::EnumeratePresentationIntervals

The final enumeration function is also called from the EnumerateDeviceOptions function. The EnumeratePresentationIntervals function fills the CD3DEnumDeviceOptions presentation interval array with options available for a given device option set. An array containing all D3DPRESENT_INTERVAL options is declared at the top of the CD3DInitialize.cpp file:

```
const ULONG     PresentIntervalCount    = 6;
const ULONG     PresentIntervals[6]     = { D3DPRESENT_INTERVAL_IMMEDIATE,
                                            D3DPRESENT_INTERVAL_DEFAULT,
                                            D3DPRESENT_INTERVAL_ONE,
                                            D3DPRESENT_INTERVAL_TWO,
                                            D3DPRESENT_INTERVAL_THREE,
                                            D3DPRESENT_INTERVAL_FOUR };
```

This function tests the **PresentationInterval** field of the **D3DCAPS9** structure to determine supported presentation intervals and adds them to the array. The virtual function ValidatePresentInterval can be overridden to reject undesirable presentation interval options.

The following code is fairly obvious and should require no more explanation.

```
HRESULT CD3DInitialize::EnumeratePresentIntervals( CD3DEnumDeviceOptions * pDeviceOptions )
{
    ULONG i, Interval;

    try
    {
        // Loop through each presentation interval
        for ( i = 0; i < PresentIntervalCount; i++ )
        {
            // Store for easy access
            Interval = PresentIntervals[i];

            // If device is windowed, skip anything above ONE
            if ( pDeviceOptions->Windowed )
            {
                if ( Interval == D3DPRESENT_INTERVAL_TWO   ||
                     Interval == D3DPRESENT_INTERVAL_THREE ||
                     Interval == D3DPRESENT_INTERVAL_FOUR ) continue;

            } // End if Windowed

            // DEFAULT is always available, others must be tested
            if ( Interval == D3DPRESENT_INTERVAL_DEFAULT )
            {
                pDeviceOptions->PresentIntervals.push_back( Interval );
                continue;

            } // Always add 'Default'

            // Supported by the device options combo ?
            if ( pDeviceOptions->Caps.PresentationIntervals & Interval )
            {
                if ( ValidatePresentInterval( Interval ) )
                    pDeviceOptions->PresentIntervals.push_back( Interval );

            } // End if Supported

        } // Next Interval Type

    } // End try Block

    catch ( ... ) { return E_OUTOFMEMORY; }

    // Success ?
    return ( pDeviceOptions->PresentIntervals.size() == 0 ) ? E_ABORT : S_OK;
}
```

Enumeration Complete

CD3DInitialize::Enumerate() initiates the process we have just examined. When the function returns, the CD3DInitialize database has been constructed and ready for use. The next step is to determine the best windowed and fullscreen modes available for device creation.

CD3DInitialize::FindBestWindowedMode

This function takes an empty CD3DSettings class and fills it with a default set of values that can be passed into the CD3DInitialize::CreateDisplay function to create a windowed device object. The strategy is straightforward since the application cannot change the adapter format (because it is currently being used by the desktop). The function loops through every device on every adapter iterating the device options array and trying to find a device option set that has the following properties:

- The adapter format matches the current display format (this is non-negotiable)
- A HAL device is preferable (unless we pass bRequireRef=TRUE as the third parameter)
- A back buffer format matching the adapter format is preferable (not essential if no match found)

Once the setting is returned from this function, the CreateDisplay member function is called passing in a width and a height. This will be the width and height of the application window and frame buffer. The application also has the option of passing two booleans into the function to request only a HAL or a REF device. In our current application we will not utilize these parameters. This indicates a willingness to use a REF device if that is all that is available on the current system.

```
bool CD3DInitialize::FindBestWindowedMode( CD3DSettings & D3DSettings, bool bRequireHAL,
bool bRequireREF )
{
    ULONG  i, j, k;
    D3DDISPLAYMODE             DisplayMode;
    CD3DEnumAdapter            *pBestAdapter = NULL;
    CD3DEnumDevice             *pBestDevice  = NULL;
    CD3DEnumDeviceOptions      *pBestOptions = NULL;
    CD3DSettings::Settings     *pSettings    = NULL;

    // Retrieve the primary adapters display mode.
    m_pD3D->GetAdapterDisplayMode( D3DADAPTER_DEFAULT, &DisplayMode);
```

The first step is determining the current adapter display mode and then to loop through every enumerated adapter. The **GetAdpaterCount** function returns the number of adapters that were stored in the adapter array at enumeration time.

```
    // Loop through each adapter
    for( i = 0; i < GetAdapterCount(); i++ )
    {
        CD3DEnumAdapter * pAdapter = m_vpAdapters[ i ];
```

For each adapter, we need to loop through each of its devices:

```
        // Loop through each device
        for( j = 0; j < pAdapter->Devices.size(); j++ )
        {
            CD3DEnumDevice * pDevice = pAdapter->Devices[ j ];
```

If this device is not a HAL device and we have specified that we require a HAL device, then we skip this device:

```
            // Skip if this is not of the required type
            if ( bRequireHAL && pDevice->DeviceType != D3DDEVTYPE_HAL ) continue;
```

If this device type is not a REF device and we have specified that we require a REF device, the same logic holds:

```
            if ( bRequireREF && pDevice->DeviceType != D3DDEVTYPE_REF ) continue;
```

At this point, this device might be suitable. We need to loop through each of the device options determine whether we can find one with a matching backbuffer and adapter mode format. This ensures that the front buffer and back buffer share the same format and speeds up scene presentation.

```
            // Loop through each option set
            for ( k = 0; k < pDevice->Options.size(); k++ )
            {
                CD3DEnumDeviceOptions * pOptions = pDevice->Options[ k ];

                // Determine if back buffer format matches adapter
                bool MatchedBB = (pOptions->BackBufferFormat == pOptions->AdapterFormat );

                // Skip if this is not windowed, and formats don't match
                if (!pOptions->Windowed) continue;
                if ( pOptions->AdapterFormat != DisplayMode.Format) continue;
```

We skip this device option if it is not a windowed option or if its adapter format is not equal to the display format the adapter is currently using.

At this point we store this mode as the best mode found so far if any of the following is true:

- No options have yet been found.
- If it is a HAL device with a matching back buffer/adapter format.
- If it is more optimal than the option stored previously.

```
         // If we haven't found a compatible option set yet, or if this set
         // is better (because it's HAL / formats match better) then save it.
         if( pBestOptions == NULL ||
                                  pOptions->DeviceType == D3DDEVTYPE_HAL && MatchedBB ) ||
            (pBestOptions->DeviceType != D3DDEVTYPE_HAL &&
             pOptions->DeviceType==3DDEVTYPE_HAL) )
        {
                    // Store best so far
                    pBestAdapter = pAdapter;
                    pBestDevice  = pDevice;
                    pBestOptions = pOptions;
```

If the current option is a HAL device with a matched backbuffer/adapter format, then we have found what we are looking for and we can exit the loop.

```
              if ( pOptions->DeviceType == D3DDEVTYPE_HAL && MatchedBB )
              {
                 // This windowed device option looks great -- take it
                 goto EndWindowedDeviceOptionSearch;
              }
         } // End if not a better match
       } // Next Option Set
    } // Next Device Type
 } // Next Adapter

EndWindowedDeviceOptionSearch:

    if ( pBestOptions == NULL ) return false;
```

If we get here and no best option has been found, then we are left with no choice but to conclude that a suitable windowed mode is not available. This is unlikely to happen. If it did happen, you may have forgotten to call Enumerate prior to entering this function.

If a best match is found then the details will be copied into the CD3DSettings object passed into the function. This object can then be used to create the final device object. This can be done manually or by calling the CD3DInitialize::CreateDisplay function.

```
    // Fill out passed settings details
    D3DSettings.Windowed                  = true;
    pSettings                             = D3DSettings.GetSettings();
    pSettings->AdapterOrdinal             = pBestOptions->AdapterOrdinal;
    pSettings->DisplayMode                = DisplayMode;
    pSettings->DeviceType                 = pBestOptions->DeviceType;
    pSettings->BackBufferFormat           = pBestOptions->BackBufferFormat;
    pSettings->DepthStencilFormat         = pBestOptions->DepthFormats[ 0 ];
    pSettings->MultisampleType            = pBestOptions->MultiSampleTypes[ 0 ];
    pSettings->MultisampleQuality         = 0;
    pSettings->VertexProcessingType       = pBestOptions->VertexProcessingTypes[ 0 ];
    pSettings->PresentInterval            = pBestOptions->PresentIntervals[ 0 ];

    // We found a mode
    return true;
}
```

By default we use the first capable depth buffer format, multi-sample type, vertex processing behavior, and presentation intervals in the arrays for this device option.

CD3DInitialize::FindBestFullscreenMode

The basic aim of this function is the same as the last. The application will pass a CD3DSettings object to be filled with a set of device creation settings. Unlike the previous function, we must pass in a D3DISPLAYMODE structure that specifies the desired width, height, refresh rate, and pixel format. The function will return as close a match as possible if the desired choices are not directly supported by any of the system adapters.

If any of the fields of the D3DDISPLAYMODE structure are zero or if the format field is set to D3DFMT_UNKNOWN, then the function will try to return a device option set which matches the current desktop display mode for that field. As a reminder, here is the D3DDISPLAYMODE once again:

```
typedef struct _D3DDISPLAYMODE
{
        UINT  Width;
        UINT  Height;
        UINT  RefreshRate;
        D3DFORMAT  Format;
} D3DDISPLAYMODE;
```

The rules of the function for the four structure members are as follows:

- If we pass a non-zero width, the function will try to return a fullscreen device option set which matches that width. If the width is zero, the function will try to return a fullscreen device option set where the width is equal to that of the current desktop video mode.
- If we pass a non-zero height, the function will try to return a fullscreen device option set which matches the passed height. If the height is zero, the function will try to return a fullscreen device option set with a height that matches that of the current desktop display mode.
- If the refresh rate is non-zero, the function will try to find a fullscreen device option set with the passed refresh rate. If this field is zero, the function will try to return a fullscreen device option set with a refresh rate equal to that of the current desktop display mode.
- If Format is a valid adapter format supported by DirectX Graphics, then the function will try to return a fullscreen device option set with this pixel format and color depth. If we pass `D3DFMT_UNKNOWN`, then the function will try to return a fullscreen device option set that matches the format of the current desktop display mode.

If the application was not particular about the current display mode it could run using the display mode the user has chosen for the desktop like so:

```
D3DDISPLAYMODE Mode;
Mode.Width          = 0;
Mode.Height         = 0;
```

```
Mode.RefreshRate    = 0;
Mode.Format         = D3DFMT_UNKNOWN;

pInitialize->FindBestFullscreenMode( &MyD3Dsettings , &Mode);
```

The fullscreen option set returned will match the current desktop display mode provided that it is supported by one of the devices on the system for fullscreen mode. If this is not the case, a closest match will be used.

```
bool CD3DInitialize::FindBestFullscreenMode(CD3DSettings & D3DSettings,
                                            D3DDISPLAYMODE * pMatchMode,
                                            bool bRequireHAL, bool bRequireREF )
{
    // For fullscreen, default to first HAL option that supports the current desktop
    // display mode, or any display mode if HAL is not compatible with the desktop mode, or
    // non-HAL if no HAL is available

    ULONG                    i, j, k;
    D3DDISPLAYMODE           AdapterDisplayMode;
    D3DDISPLAYMODE           BestAdapterDisplayMode;
    D3DDISPLAYMODE           BestDisplayMode;
    CD3DEnumAdapter         *pBestAdapter = NULL;
    CD3DEnumDevice          *pBestDevice  = NULL;
    CD3DEnumDeviceOptions   *pBestOptions = NULL;
    CD3DSettings::Settings  *pSettings    = NULL;

    BestAdapterDisplayMode.Width  = 0;
    BestAdapterDisplayMode.Height = 0;
    BestAdapterDisplayMode.Format = D3DFMT_UNKNOWN;
    BestAdapterDisplayMode.RefreshRate = 0;

    // Loop through each adapter
    for( i = 0; i < GetAdapterCount(); i++ )
    {
        CD3DEnumAdapter * pAdapter = m_vpAdapters[ i ];

        // Retrieve the desktop display mode
        m_pD3D->GetAdapterDisplayMode( pAdapter->Ordinal, &AdapterDisplayMode );

        // If any settings were passed, overwrite to test for matches
        if ( pMatchMode )
        {
            if ( pMatchMode->Width  != 0 ) AdapterDisplayMode.Width  = pMatchMode->Width;
            if ( pMatchMode->Height != 0 ) AdapterDisplayMode.Height = pMatchMode->Height;
            if ( pMatchMode->Format != D3DFMT_UNKNOWN )
               AdapterDisplayMode.Format = pMatchMode->Format;
            if ( pMatchMode->RefreshRate != 0 )
               AdapterDisplayMode.RefreshRate = pMatchMode->RefreshRate;

        } // End if match mode passed
```

A local D3DISPLAYMODE structure is constructed for the desired display mode and will be used to search the database for a match. Notice that in the **if(pMatchMode)** code block we copy over the fields of the passed D3DDISPLAYMODE structure unless one of the fields is zero. In that case, we copy the information from the adapter current display mode. This allows us to leave certain fields in

the passed D3DDISPLAYMODE structure as zero and forces the function to use the current display mode for those values.

The function now needs to test every adapter and every device on that adapter for an option set that matches the display mode of the local AdapterDisplayMode structure.

```
        // Loop through each device
        for( j = 0; j < pAdapter->Devices.size(); j++ )
        {
            CD3DEnumDevice * pDevice = pAdapter->Devices[ j ];
```

As with the previous function, if the application has specified an explicit requirement for either a HAL or REF device, appropriate steps are taken:

```
            // Skip if this is not of the required type
            if ( bRequireHAL && pDevice->DeviceType != D3DDEVTYPE_HAL ) continue;
            if ( bRequireREF && pDevice->DeviceType != D3DDEVTYPE_REF ) continue;
```

Now that we have a device that might be valid, we test all of its option sets to find one that is best suited to the requested format. We record whether the current option set being tested has a matching adapter and backbuffer format since this is an optimal arrangement. We also record whether the adapter format of the option set exactly matches the adapter format that we are looking for. The name of the variable *MatchedDesktop* is potentially a little misleading. It is not set to true if the option set adapter mode matches the current desktop format (as its name suggests). Instead, it is set to true if the option set format matches the format we are looking for. But if we did not pass in a specific format then the *AdapterDisplayMode.Format* member will contain the desktop format by default. For obvious reasons, we skip windowed mode option sets.

```
            // Loop through each option set
            for ( k = 0; k < pDevice->Options.size(); k++ )
            {
                CD3DEnumDeviceOptions * pOptions = pDevice->Options[ k ];

                // Determine if back buffer format matches adapter
                bool MatchedBB = (pOptions->BackBufferFormat == pOptions->AdapterFormat );
                bool MatchedDesktop = \
                                    (pOptions->AdapterFormat == AdapterDisplayMode.Format);

                // Skip if this is not fullscreen
                if ( pOptions->Windowed ) continue;
```

If we get this far, then we have found a potential candidate option set for a fullscreen device. The next step is to determine whether it is in fact the best set found thus far. If there is no previously stored best option set, then this one automatically becomes the new best option set. If there is an existing best option set, but it is not from a HAL device and the current one is, then we make this new option set the best set. If they are both HAL sets, but the previously stored best option set does not precisely match the requested format and the new one does, then this becomes the new best option set. Finally, if this new option set is a HAL device and it matches our requested format and it also has a matching adapter/backbuffer format combination, then this is an ultimate match and we can stop our search.

```
                // If we haven't found a compatible option set yet, or if this set
                // is better (because it's HAL / formats match better) then save it.
                if ( pBestOptions == NULL ||
                     (pBestOptions->DeviceType != D3DDEVTYPE_HAL &&
                      pDevice->DeviceType == D3DDEVTYPE_HAL ) ||
                     (pOptions->DeviceType == D3DDEVTYPE_HAL  &&
                      pBestOptions->AdapterFormat != AdapterDisplayMode.Format &&
                      MatchedDesktop)  ||
                     (pOptions->DeviceType == D3DDEVTYPE_HAL &&
                      MatchedDesktop && MatchedBB))
                {
                    // Store best so far
                    BestAdapterDisplayMode = AdapterDisplayMode;
                    pBestAdapter = pAdapter;
                    pBestDevice  = pDevice;
                    pBestOptions = pOptions;

                    if ( pOptions->DeviceType == D3DDEVTYPE_HAL &&
                         MatchedDesktop && MatchedBB )
                    {
                        // This fullscreen device option looks great -- take it
                        goto EndFullscreenDeviceOptionSearch;
                    }

                } // End if not a better match

            } // Next Option Set

        } // Next Device Type

    } // Next Adapter

EndFullscreenDeviceOptionSearch:

    if ( pBestOptions == NULL) return false;
```

At this point we hopefully have found a matching option set. Even if it was not an exact match, we should at least have an option set that comes fairly close. We copied all adapter and device information into the local pBestAdapter, pBestDevice, and pBestOptions variables.

We must still loop through all of the best adapter display modes (stored in a separate array within the CD3DEnumAdpater class) and find a display mode that matches our new best format. It also has to match the width, height, and refresh rate passed into the function. We will store the results in a local D3DISPLAYMODE structure called BestDisplayMode.

```
    // Need to find a display mode on the best adapter
    // that uses pBestOptions->AdapterFormat
    // and is as close to BestAdapterDisplayMode's res as possible
    BestDisplayMode.Width       = 0;
    BestDisplayMode.Height      = 0;
    BestDisplayMode.Format      = D3DFMT_UNKNOWN;
    BestDisplayMode.RefreshRate = 0;
```

Loop through each of the adapters display mode and reject any that do not match the adapter format of our previously found best option set.

```
    // Loop through valid display modes
    for( i = 0; i < pBestAdapter->Modes.size(); i++ )
    {
        D3DDISPLAYMODE Mode = pBestAdapter->Modes[ i ];

        // Skip if it doesn't match our best format
        if( Mode.Format != pBestOptions->AdapterFormat ) continue;
```

This display mode has a matching format. If it is a perfect match, then we can break from this loop because we need to look no further.

```
        // Determine how good a match this is
        if( Mode.Width == BestAdapterDisplayMode.Width &&
            Mode.Height == BestAdapterDisplayMode.Height &&
            Mode.RefreshRate == BestAdapterDisplayMode.RefreshRate )
        {
            // found a perfect match, so stop
            BestDisplayMode = Mode;
            break;

        } // End if Perfect Match
```

If we get to this point, then the display mode is not a perfect match but may be better than any we have found in previous iterations of the loop. The next step is to check for a match with everything except the refresh rate. If this test passes, then the display mode has matching width, height, and adapter format but a different refresh rate. This is a decent match and we store the current display mode as the best so far and continue the loop.

```
        else if( Mode.Width == BestAdapterDisplayMode.Width &&
                 Mode.Height == BestAdapterDisplayMode.Height &&
                 Mode.RefreshRate > BestDisplayMode.RefreshRate )
        {
            // refresh rate doesn't match, but width/height match, so keep this
            // and keep looking
            BestDisplayMode = Mode;
        }
```

At this point, we test to see if the width of the display mode matches and store it as the best found so far if it does. This indicates that the display mode returned may have a different height resolution and refresh rate, but will have the desired width and adapter format.

```
        else if( Mode.Width == BestAdapterDisplayMode.Width )
        {
            // width matches, so keep this and keep looking
            BestDisplayMode = Mode;
        }
```

If we reach this point, then this display mode is not a very good match at all since only the format matches. If this is the case, then we will store only the current display mode as the best found so far.

```
        else if( BestDisplayMode.Width == 0 )
        {
            // we don't have anything better yet, so keep this and keep looking
            BestDisplayMode = Mode;

        } // End if

    } // Next Mode
```

Now the job is done and we have hopefully found at least a decent match. We copy the information into the CD3DSettings structure passed into the function and return control back to the caller.

```
    // Fill out passed settings details
    D3DSettings.Windowed               = false;
    pSettings                          = D3DSettings.GetSettings();
    pSettings->AdapterOrdinal          = pBestOptions->AdapterOrdinal;
    pSettings->DisplayMode             = BestDisplayMode;
    pSettings->DeviceType              = pBestOptions->DeviceType;
    pSettings->BackBufferFormat        = pBestOptions->BackBufferFormat;
    pSettings->DepthStencilFormat      = pBestOptions->DepthFormats[ 0 ];
    pSettings->MultisampleType         = pBestOptions->MultiSampleTypes[ 0 ];
    pSettings->MultisampleQuality      = 0;
    pSettings->VertexProcessingType    = pBestOptions->VertexProcessingTypes[ 0 ];
    pSettings->PresentInterval         = pBestOptions->PresentIntervals[ 0 ];

    // Success!
    return true;
}
```

This was certainly a lot of information to absorb. Please take time to study the source code so that you can make adjustments down the road to meet your own application needs.

There is still one initialization phase left to discuss. Phase 1 enumerated the devices on the system. Phase 2 used the FindBestXX functions to search the enumerated database for a compatible set of device settings. Phase 3 will now use these settings to create the device and optionally create an application window.

CD3DInitialize::CreateDisplay

The CreateDisplay function is used to initialize the Direct3D device and to optionally create an application window. If the input HWND parameter is NULL, then the function will create the application window as well as the device. If this is the desired behavior, then a WNDPROC function will also be required. If you already have an application window, then simply pass in the HWND of your window and it will create the device only. Note that the function may need to alter some of the attributes of your window to make it work with a fullscreen device (such as removing the menu and moving its origin to screen coordinate <0, 0>)

The first section of code creates the window if the HWND parameter was set to NULL. It also tests to see if we are creating a fullscreen device. If so, then it sets the width and height of the window to that

of the requested fullscreen display mode. If window creation fails, then the function will exit with a failure notification.

```
HRESULT CD3DInitialize::CreateDisplay( CD3DSettings& D3DSettings, ULONG Flags, HWND hWnd,
                                       WNDPROC pWndProc, LPCTSTR Title, ULONG Width,
                                       ULONG Height, LPVOID lParam )
{
    ULONG CreateFlags                   = 0;
    CD3DSettings::Settings *pSettings   = D3DSettings.GetSettings();

    if ( !hWnd )
    {
        // Register the new windows window class.
        WNDCLASS wc;
        wc.style              = CS_BYTEALIGNCLIENT | CS_HREDRAW | CS_VREDRAW;
        wc.lpfnWndProc        = pWndProc;
        wc.cbClsExtra         = 0;
        wc.cbWndExtra         = 0;
        wc.hInstance          = (HINSTANCE)GetModuleHandle(NULL);
        wc.hIcon              = NULL;
        wc.hCursor            = LoadCursor(NULL, IDC_ARROW);
        wc.hbrBackground      = (HBRUSH )GetStockObject(WHITE_BRUSH);
        wc.lpszMenuName       = NULL;
        wc.lpszClassName      = Title;
        RegisterClass(&wc);

        ULONG Left  = CW_USEDEFAULT, Top = CW_USEDEFAULT;
        ULONG Style = WS_OVERLAPPEDWINDOW;

        // Create the rendering window
        if ( !D3DSettings.Windowed )
        {
            Left   = 0; Top = 0;
            Width  = pSettings->DisplayMode.Width;
            Height = pSettings->DisplayMode.Height;
            Style  = WS_VISIBLE | WS_POPUP;

        } // End if Fullscreen

        // Create the window
        m_hWnd = CreateWindow( Title, Title, Style, Left, Top, Width, Height,
                               NULL, NULL, wc.hInstance, lParam );

        // Bail on error
        if (!m_hWnd) return E_FAIL;

    } // End if no Window Passed
```

The next block of code examines the scenarios where the HWND parameter is not set to NULL. This window will be used as the device window.

```
    else
    {
        // Store HWND
        m_hWnd = hWnd;
```

```
        // Setup styles based on windowed / fullscreen mode
        if ( !D3DSettings.Windowed )
        {
            SetMenu( m_hWnd, NULL );
            SetWindowLong( m_hWnd, GWL_STYLE, WS_VISIBLE | WS_POPUP );
            SetWindowPos( m_hWnd, NULL, 0, 0, pSettings->DisplayMode.Width,
                          pSettings->DisplayMode.Height, SWP_NOZORDER );

        } // End if Fullscreen
        else
        {
            RECT rc;

            // Get the windows client rectangle
            GetWindowRect( hWnd, &rc );

            // Setup the window properties
            SetWindowLong( m_hWnd, GWL_STYLE, WS_OVERLAPPEDWINDOW );
            SetWindowPos( hWnd, HWND_NOTOPMOST, rc.left, rc.top,
                         (rc.right - rc.left), (rc.bottom - rc.top),
                          SWP_NOACTIVATE | SWP_SHOWWINDOW );

        } // End if Windowed

    } // End if window passed
```

The next task is to fill the D3DPRESENT_PARAMETERS structure with the settings found in the input CD3DSettings object. The following code uses a helper function called BuildPresentParameters to copy all of the fields from the CD3DSettings object into the D3DPRESENT_PARAMETERS structure.

```
    // Build our present parameters
    D3DPRESENT_PARAMETERS d3dpp = BuildPresentParameters( D3DSettings );
```

At this point, we have a D3DPRESENT_PARAMETERS structure and a window to use as the device window. What remains is to determine whether the CD3DSettings structure requests the use of software, hardware, or mixed vertex processing. Once done, we create the device and store its pointer in the CD3DInitialize member variable.

```
    // Build our creation flags
    if ( pSettings->VertexProcessingType == PURE_HARDWARE_VP )
        CreateFlags |= D3DCREATE_PUREDEVICE | D3DCREATE_HARDWARE_VERTEXPROCESSING;
    else if ( pSettings->VertexProcessingType == HARDWARE_VP )
        CreateFlags |= D3DCREATE_HARDWARE_VERTEXPROCESSING;
    else if ( pSettings->VertexProcessingType == MIXED_VP )
        CreateFlags |= D3DCREATE_MIXED_VERTEXPROCESSING;
    else if ( pSettings->VertexProcessingType == SOFTWARE_VP )
        CreateFlags |= D3DCREATE_SOFTWARE_VERTEXPROCESSING;

    // Create the device
    m_pD3DDevice = NULL;
    HRESULT hRet = m_pD3D->CreateDevice( pSettings->AdapterOrdinal, pSettings->DeviceType,
                                         m_hWnd, CreateFlags, &d3dpp, &m_pD3DDevice );
```

```
    // Did the creation fail ?
    if ( FAILED( hRet ) )
    {
        if ( m_pD3DDevice ) m_pD3DDevice->Release();
        m_pD3DDevice = NULL;
        return hRet;
    } // End if failed

    // Success
    return S_OK;
}
```

Assuming that the function was successful, our application can now call CD3DInitialize::GetDirect3DDevice to retrieve a pointer to the Direct3D device interface. At that point the application can let the CD3DInitialize object go out of scope or delete it if it was allocated on the heap.

For completeness, the code to the BuildPresentParameters helper function is shown below:

```
D3DPRESENT_PARAMETERS CD3DInitialize::BuildPresentParameters(CD3DSettings& D3DSettings,
                                                             ULONG Flags)
{
    D3DPRESENT_PARAMETERS   d3dpp;
    CD3DSettings::Settings *pSettings = D3DSettings.GetSettings();

    ZeroMemory ( &d3dpp, sizeof(D3DPRESENT_PARAMETERS) );

    // Fill out our common present parameters
    d3dpp.BackBufferCount           = 1;
    d3dpp.BackBufferFormat          = pSettings->BackBufferFormat;
    d3dpp.Windowed                  = D3DSettings.Windowed;
    d3dpp.MultiSampleType           = pSettings->MultisampleType;
    d3dpp.MultiSampleQuality        = pSettings->MultisampleQuality;
    d3dpp.EnableAutoDepthStencil    = TRUE;
    d3dpp.AutoDepthStencilFormat    = pSettings->DepthStencilFormat;
    d3dpp.PresentationInterval      = pSettings->PresentInterval;
    d3dpp.Flags                     = D3DPRESENTFLAG_DISCARD_DEPTHSTENCIL | Flags;
    d3dpp.SwapEffect                = D3DSWAPEFFECT_DISCARD;

    // Is this fullscreen ?
    if ( !d3dpp.Windowed )
    {
        d3dpp.FullScreen_RefreshRateInHz = pSettings->DisplayMode.RefreshRate;
        d3dpp.BackBufferWidth            = pSettings->DisplayMode.Width;
        d3dpp.BackBufferHeight           = pSettings->DisplayMode.Height;

    } // End if fullscreen

    // Success
    return d3dpp;
}
```

User Config Dialog

To make the use of the enumeration concepts discussed in this lesson a little easier, you will find an extra class called CD3DSettingsDlg included with the material. The source can be found in CSettingDlg.cpp and CSettingDlg.h. The class provides an application with the ability to let the user select the device option set they wish to use to run the application. This dialog class is used in Lab Project 2.2 in the CGameApp::CreateDisplay function as follows:

```
    D3DDISPLAYMODE  MatchMode;
    CD3DSettingsDlg SettingsDlg;
    CMyD3DInit      Initialize;

    // First of all create our D3D Object (This is needed by the enumeration etc)
    m_pD3D = Direct3DCreate9( D3D_SDK_VERSION );

    // Enumerate the system adapters/devices
    Initialize.Enumerate( m_pD3D )

    // Attempt to find a good default fullscreen set
    MatchMode.Width       = 640;
    MatchMode.Height      = 480;
    MatchMode.Format      = D3DFMT_UNKNOWN;
    MatchMode.RefreshRate = 0;
    Initialize.FindBestFullscreenMode( m_D3DSettings, &MatchMode );

    // Attempt to find a good default windowed set
    Initialize.FindBestWindowedMode( m_D3DSettings );
```

At this point, m_D3DSettings contains settings for a fullscreen and windowed mode device. Because FindBestWindowedMode was called last, the m_D3DSetting::Windowed boolean will be set to true. This indicates a desire to use the windowed option set.

Next, we see some new code that passes the m_D3Dsettings object into the CSettingsDlg::ShowDialog function. This will display the configuration dialog box and initializes all of its controls to display the options passed in the m_D3Dsettings object. The m_D3DSettings object is only being used to provide a set of default selections for the dialog box when it first opens.

```
    int RetCode = SettingsDlg.ShowDialog( &Initialize, &m_D3DSettings );
```

After the user has made their selections and press the ok button, the settings will be stored inside the CD3DSettingsDlg class in its own CD3DSetting structure. The application can now retrieve the user options and use them to create the device object:

```
    m_D3DSettings = SettingsDlg.GetD3DSettings();

    Initialize.CreateDisplay(m_D3DSettings,0,NULL,
                             StaticWndProc,WindowTitle,Width,Height,this))
    // Retrieve created items
    m_pD3DDevice   = Initialize.GetDirect3DDevice( );
    m_hWnd         = Initialize.GetHWND( );
```

Chapter 2 Appendix A

The Projection Matrix Z-Buffer Requirements

This brief appendix discusses why Z-Buffers do not distribute depth values linearly as well as possible ways to resolve certain Z-Buffer artifacts.

DirectX Graphics requires that after the 4D vector is returned from the projection matrix, and after it has divided the ***x***, ***y*** and ***z*** values of this vector by ***w***, ($\frac{x}{w}\frac{y}{w}$ & $\frac{z}{w}$), the ***z*** value to be in the range 0.0 to 1.0. In this case 0.0 would describe a point on the near clip plane and 1.0 would describe a point on the far clip plane. Any *Z* value between the 0.0 and 1.0 range is considered to be within the view frustum (provided it is within the FOV in both the x and y dimensions). The application needs to ensure that the third column in our projection matrix is such that, when DirectX Graphics divides it by ***w***, values are returned in the 0.0 to 1.0 range for any point inside the frustum.

As mentioned in chapter 1, the projection matrix does not actually project points. The divide by *w* is performed on the vector that is returned from the projection matrix multiplication which produces the 2D projection. This same logic holds true for the Z value. This value should not end up in the 0.0 to 1.0 range until the divide by *w* takes place. The third column of the projection matrix must be set up to map the input vector ***z*** component to some other space, so that it ends up in the 0.0 to 1.0 range after the divide by *w*. Also, recall that the ***z*** value of the input vector is copied over to the ***w*** value of the output vector.

For clarity sake, in the examples in this section we will set the first and second columns of the projection matrix to x and y identity columns. We will concentrate only on the third column values.

The third column of the projection matrix has only two rows (3 and 4) that can be used to produce a value based on the input z component. We will labels these as 'a' and 'b' in the following matrix. Our goal will be to find values to fill in here that satisfy the specified requirements. It should be immediately clear that the first and second rows of the third column would not be of much value since they would factor the input vectors x and y components in to the resulting z value and this is not what we want.

Projection Matrix

$$V = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \times M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & 1 \\ 0 & 0 & b & 0 \end{bmatrix} = P\begin{bmatrix} X = x & Y = y & Z = \text{ZBufferInfo} & W = z \end{bmatrix}$$

We know that after projection ***W*** is equal to the ***z*** component of the input vector. So we know that the ***Z*** component of the output vector needs to be a number such that dividing by ***W*** will scale it into the

0.0 to 1.0 range when the input vector ***z*** component is between the near and far plane in view space. Since ***W = z*** we will need a new Z value such that ***Z / z =0.0 to 1.0*** when the vector is inside the view frustum. (***Z*** = output Z || ***z*** = input Z)

Certainly we cannot simply copy the input vector ***z*** value into the output vector ***Z*** value. Otherwise ***W=Z*** would always be true in the output vector. When DirectX Graphics does the divide by ***w*** it would calculate the Z-Buffer value like so: $\frac{Z}{W}$ which would be equal to: $\frac{Z}{Z} or \frac{W}{W}$ which will always result in a value of ***1.0***. All points rendered would have the same Z value (at the very back of the depth buffer) and the Z-Buffer would be useless.

So we need to calculate a new ***Z*** value based on the input ***z*** value, but not directly proportional to it. It is also important to realize that if the far plane was 100 units from the camera and the near plane was 10 units from the camera, the Z relationship to the camera from any point is not that same as the Z relationship with the Z-Buffer. The Z-Buffer is only interested in values that fall between the near and far planes. If a vector has a ***z*** value of 20 for example, it means that it is 20 units from the camera. However this does not mean that we want to write a value into the Z-Buffer that is equivalent in percentage terms (20% from the near plane). Z-Buffer space starts at the near plane. This means the Z view space points of 10.0 in our example would be on the near plane and should result in a value of 0.0 because it is at the very front of near plane/far plane space. This will be what makes our ***Z*** calculation possible.

For the rest of the discussion we will assume the following conditions:

Near Plane =10.0
Far Plane =100.0

The first thing we must do to our input ***z*** value is subtract the distance to the near plane (10.0 in our example). The job of the third column of the matrix is to produce a depth value for the Z-Buffer based on the input view space z value. Any input vectors that have z values < 10 will have this distance subtracted from them. Any values that are between 0 and 10 (although technically in front of the camera in view space) will wind up in range [-10.0, 0.0] and will fail the Z-Buffer test. This has the effect of rejecting any geometry that is closer to the camera than the near plane.

There is one thing to note. Because we subtract 10 from the input ***z*** coordinate we are only interested in ***z*** values between 0-90 (Originally we were interested only in ranging between 10.0 and 100.0 because these were the view space values between the near and far planes). Subtracting 10 from the ***z*** value takes the value into what we might call near plane/far plane space. So a ***z*** value of 90 was originally 100 units from the camera in camera space. We need to make sure that any input ***z*** value is multiplied in such a way that a point at 100 in camera space (on the far plane) results in an output ***Z*** value of 100 (***Z=W***) from the projection matrix as well.

This may sound simple at first but keep in mind that we have already subtracted the near plane distance from the ***z*** values. Any input ***z*** value that was previously equal to 100.0 would have been reduced to a value of 90.0. This means that this point is on the far plane and should ultimately leave the projection matrix equal to W. When this is the case it creates a Z-Buffer value of Z/W = 1.0. In our example this means that a ***z*** value of 100 in view space would have been reduced to 90 after near plane subtraction and should now be multiplied by some number such that it would make it equal to 100 again.

The following formula satisfies our needs:

$$Z = \frac{FarPlane}{(FarPlane - NearPlane)}$$

This formula creates a value that we can multiply our input z value with (after near plane subtraction) and will scale the z value in such a way that a value of 90 in our example will result in an output z value of 100 again. We have found a way to map the 0-90 range of values back to the 0-100 range of values.

$$Z = z - nearplane \times \left(\frac{FarPlane}{(FarPlane - NearPlane} \right)$$

$$Z = z - 10.0 \times \left(\frac{100.0}{90} = 1.11111111 \right)$$

$$Z = (z - 10.0) \times 1.11111111$$

To test this approach, let us plug in some values and see the results. One obvious value to test for compliance is an initial input view space z value of 100.0. We already know that it is positioned on the far plane and that it should eventually end up being converted to a maximum z buffer value of 1.0.

$$\begin{aligned} z &= 100.0 \\ Z &= (100 - 10) = 90 \\ Z &= 90 \times 1.11111111 = 99.9999999 \end{aligned}$$

When DirectX Graphics does the ***divide by w,*** the final Z-Buffer depth value (***ZB***) is:

$$ZB = \frac{Z}{W} = \frac{99.99999999}{100.0} = 0.99999999$$

Allowing for floating point precision we can see that this works perfectly.

Let us check a point on the near plane next. Because the near plane in our examples is positioned at 10.0 from the camera in view space we know that a z value of 10.0 should be on the near plane and mapped at the very front of the depth buffer (0.0 after the ***divide by w***):

$$Z = (10.0 - 10.0 = 0) \times 1.11111111 = 0.0$$

$$ZB = \frac{Z}{W} = \frac{0.0}{10} = 0.0$$

It should be obvious to you that any view space value closer to the camera than the near plane (8.0 in view space for example) would be mapped to a final Z-Buffer value that was < 0.0:

$$Z = (8.0 - 10.0 = -2) \times 1.11111111 = -2.22222222$$

$$ZB = \frac{Z}{W} = \frac{-2.22222222}{8} = -0.277777777$$

This result would be rejected.

Now that we know what we want to do, the next step is figuring out how to do this in a matrix.

$$V = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \times M = \overset{\text{Projection Matrix}}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & 1 \\ 0 & 0 & b & 0 \end{bmatrix}} = P\begin{bmatrix} X = x & Y = y & Z = \text{ZBufferInfo} & W = z \end{bmatrix}$$

The first thing we need to do is subtract the near plane distance from the input z value. This presents an immediate problem because as we discussed in chapter 1 (while discussing translations), the only row that we can use to perform addition or subtraction is the fourth row. Recall that this is because the input value of **w** will always equal 1.0. So the only place where we can force the subtraction of the near plane into our linear transformation is in element ***b*** in the above matrix. The problem is that ***b*** is in the last row and we would no longer have a means tom complete the transformation and perform the multiply (by 1.11111111 in our example).

The solution is to reverse the order in which we do the above calculation. First we multiply the input z value by the ratio (1.11111111 in our example). To do this we can just store our ratio value in element ***a*** in the above matrix. This will (so far) create an output Z value:

Z=Vx*0 + Vy*0 + Vz****a*** + Vw****b***

If we put the ratio (1.11111111) into element ***a***:

Z=Vz*1.11111111 + 1****b***

The z value has been scaled first. Using the ***w=1*** assumption we can simply put a negative value into the ***b*** element to subtract the near plane distance.

You would be forgiven for thinking that all we have to do in our example is put –10 into element ***b***. However this is not the case because we have already scaled the input z value by our ratio. Because we did not subtract the near plane distance first it means that this distance (10.0 in our example) has also been multiplied by 1.11111111. Therefore we need to subtract this amount using the ratio again. Instead of subtracting 10.0 we need to subtract 10.0*1.11111111:

$$\textbf{Ratio}=\frac{FarPlane}{(FarPlane-NearPlane)}=\frac{100.0}{90}=1.11111111$$

Projection Matrix

$$V=\begin{bmatrix}x & y & z & 1\end{bmatrix}\times M=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & Ratio & 1\\ 0 & 0 & -NearPlane*Ratio & 0\end{bmatrix}=P\begin{bmatrix}X=x & Y=y & Z=\text{ZBufferInfo} & W=z\end{bmatrix}$$

Our third column is now complete. Let us test it with a z value of 100 again just to make sure:

Z = Vz*ratio = (100*1.11111111) (=111.11111111)
+
w * (-NearPlane * Ratio) = 1 * -10*1.11111111 (= -11.11111111)

= 111.11111111- 11.11111111
= 99.99999999

When we divide by ***w*** (100 in our example) we get a Z-Buffer value:

$$ZB=\frac{Z}{W}=\frac{99.99999999}{100.00}=0.99999999$$

The Z-Buffer Is Not Linear

Because we are using the ratio to multiply the input z value, the output z value is not linearly distributed across the range of the Z-Buffer. If we had a view space ***z*** value of 55, we know that this point in view space is exactly half way between the near and far planes. This is because the near plane starts at 10 so the half way point would be 10 + (90/2=45) = 55.

If the third column produced a linear mapping with the Z-Buffer you would expect the final value to be equal to 0.5 (halfway between 0.0 and 1.0). However this is not the case:

z=55 (view space z, halfway between near and far planes)

Z= z *Ratio * 1 * (-NearPlane * Ratio)
Z= 55*1.11111111 * 1 * (-10 * 1.11111111)
Z =61.11111105 - 11.11111111
Z= 49.99999999 (Output ***Z*** from projection matrix)

An input z of 55.0 produces the output Z value of 49.99999999. When we divide by ***w***:

$$ZB = \frac{Z}{W} = \frac{49.99999999}{55.0} = 0.909090908$$

This is a bit surprising. The Z-Buffer value it calculated is right near the back in the 0.9 range. Because we are multiplying the z values by 1.11111111, there is a kind of cascade effect that brings each point in the 0 to 90 range exponentially nearer to the 0-100 range. To better see this in action, take a look at the following table. It shows the projection matrix output for view space ***z*** values in increments of 20.0.

Near plane = 10.0
Far plane = 100.0
Ratio = 1.11111111

Camera Space Z Values:

20.0	40.0	60.0	80.0	100

Z Values returned from projection matrix:

11.11111111	33.33333333	55.55555555	77.77777777	99.99999999

Can you see the pattern in the above table that indicates why the Z Buffer calculation would not be linear? At a distance of 20.0 units from the camera in view space the value is mapped to 11.11111111. The difference between the input z value and the output ***Z*** value is 8.88888888. A view space z value of 40 gets mapped to 33.33333333. The difference between these two values is only 6.66666666. You will see by following the numbers that the difference grows progressively smaller as the input ***z*** value increases until eventually the difference between the two values is zero (or nearly zero) at 100 in camera space.

Recall that the output value is divided by w and that w is equal to the input ***z*** value. As the difference between the input and output values decrease with increasing ***z*** the divide by w creates a number increasingly closer to 1.0 (because these two values grow more and more similar)

Final Z Buffer values after divide by w:

0.55555555	0.833333333	0.925925925	0.9722222222	0.99999999

We see now that the z buffer values are not linearly distributed across the range. In fact, at a distance of 10.0 from the near plane (input z=20) we have already used up half of the values (> 0.5). This indicates that 50% of the Z-Buffer precision has been used up in the first 10.0 units of near plane/far plane space. By the time we hit a distance of 50 units from the near plane we are already computing values well into the 0.9 range of the z buffer. This problem gets worse when the distance between the near and far planes increase. In a typical application, the far plane is often set quite a bit further than 100 camera space units.

This (unavoidable) non-linear mapping creates problems that have been a long time hindrance to game developers. When using Z-Buffers of 16 bits or less, you will often see artifacts (often called Z Fighting or Z Wars).

These artifacts are caused by the fact that 90% of the Z-Buffer precision is typically used up in the closest 10% of the scene. If many objects are far away from the camera, we can have a result where several points at different locations in 3D space map to the same Z-Buffer value.

Quake™ players may remember playing DM3 and camping out by the mega health in the pent courtyard. Sometimes people hiding in the enclave on the opposite side of the courtyard would appear through the wall that should have been obscuring them. This was caused by a lack of Z-Buffer precision in the Quake™ software rasterizer.

Z-Buffer artifacts are less common close to the camera because precision is adequate in that range. Note that it is actually the projection matrix that causes this problem. It is not a hardware problem. Although there is little we can do to change the mathematics, there are ways to deal with this problem:

- Using a 24 bit Z-Buffer almost always solves this problem. 24 bit Z-Buffers offer so much resolution that using 32 bits is generally considered wasteful. A 32 bit Z-Buffer would offer 4 billion possible depth values between the near and far plane. That is likely far more than we will ever need. This is the reason why the top 8 bits of a 32 bit Z-Buffer are usually reserved for stencil buffering.

- More recent hardware includes a W-Buffer. The W-Buffer uses the W component of the projection matrix output vector for the depth calculation. The W-Buffer maps much more linearly than a Z-Buffer and is excellent for getting rid of Z-Buffer artifacts. The buffer uses the same memory as the Z-Buffer and is similar in most other ways.

- Reducing the distance between the near and far plane help reduce artifacts. It is actually a lot more effective to move the near plane forward when it comes to curing artifacts but you are very limited by how far you can move the near plane before objects start getting clipped inappropriately. If you can get away with moving the near plane a bit and it does not cure the problem completely, try moving the far plane back a bit as well. The goal we are trying to achieve to get a more linear mapping is to reduce the ratio used by the projection matrix while making sure Z maps from 0.0 to 1.0 after the divide by W.

Our Projection Matrix now has a third column that looks like this:

Projection Matrix

$$\text{Ratio} = \frac{FarPlane}{FarPlane - NearPlane}$$

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & Ratio & 1 \\ 0 & 0 & -NearPlane * Ratio & 0 \end{bmatrix}$$

Chapter 2 Appendix B Render/Transform States

RenderState Type		Argument		Description
D3DRS_ZENABLE		D3DZB_TRUE , D3DZB_FALSE or D3DZB_USEW		Used to enable or disable the Z Buffer. The Z Buffer must have been created and attached to the device. D3DZB_USEW instructs the device to use w buffering. You must first check if W Buffering is available.
D3DRS_SHADEMODE		D3DSHADE_FLAT or D3DSHADE_GOURAUD		Used to set the shading policy used by the device when rendering primitives. The default is D3DSHADEMODE_GOURAUD which interpolates color stored at the vertices over the surface.
D3DRS_CULLMODE		D3DCULL_NONE, D3DCULL_CCW or D3DCULL_CW		Tells the device which winding order is to be considered for back face removal. The default is D3DCULL_CCW which states primitives with a counter clockwise winding order with respect to the camera should be culled.
D3DRS_FILLMODE		D3DFILL_POINT, D3DFILL_WIREFRAME, D3DFILL_SOLID		Tells the device how vertex lists should be rendered. The default is D3DFILL_SOLID stating that primitives should be filled with color depending on the current shade mode.
D3DRS_LIGHTING		TRUE Or FALSE		Enables or disables the devices lighting module during the transformation of vertices. The default value is TRUE. Only vertices which include a vertex normal will be lit correctly.
D3DRS_DITHERENABLE		TRUE Or FALSE		Enables or disable dithering on the device. The default value is FALSE.

Transform State Type		Argument		Description
D3DTS_WORLD		D3DXMATRIX * World		Used to set the devices **world** matrix. You pass in the address of a D3DXMATRIX containing the new world matrix you want to set.
D3DTS_VIEW		D3DXMATRIX * View		Used to set the devices **view** matrix. You pass in the address of a D3DXMATRIX containing the new view matrix you want to set.
D3DTS_PROJECTION		D3DXMATRIX * Proj		Used to set the devices projection matrix. You pass in the address of a D3DXMATRIX containing the new projection matrix you want to set.

Chapter 2 Appendix C

STL Vector Primer

The Standard Template Library (STL) is an integral part of any C++ toolset. It provides many different template classes for performing routine tasks such as memory allocation and string handling. One of the most commonly used templates is a container called a 'vector'. A vector is essentially a dynamic array. It provides easy allocation, re-allocation, and de-allocation of linearly indexed memory. While we have decided not to use vectors in place of standard arrays in most of our demo applications, we do use them to simplify working with the many different types of arrays required by the enumeration and initialization systems.

We declare a vector of any arbitrary type in the manner shown below:

```
std::vector<int> m_IntVector;
```

The first part of this line, std::, instructs the compiler that the following type is a member of the 'std' namespace. This is typically always the case with the common STL types so you could also make use of the 'using namespace std;' directive to avoid having to include the namespace explicitly.

The next portion of code declares the variable to be of type 'vector<int>'. Because this is a template, we are able to specify the type of data to be stored and managed. We could replace the 'int' in the above example with other data types, including structures or pointers.

Note: Older versions of the Microsoft STL vector implementation (such as the one provided with Visual C++ 5.0) are not strictly compliant when it comes to user defined structures as the input type. Therefore it is often preferable simply to store pointers to those structures.

The following is a short list of some of the common vector functions we will use in our applications:

Adding Items

vector::push_back accepts an object of the declared type and adds it to the end of the stored array. If the allocated memory block is not large enough to hold the new item, then the vector will grow automatically to make room:

```
int IntVar = 3;

// Add this integer value to our vector
m_IntVector.push_back( IntVar );

// Also feel free to push a constant if you wish :)
m_IntVector.push_back( 3 );
```

vector::size() returns the number of items currently stored in the vector.

A major disadvantage to adding items in this way (one by one) is that the vector may be required to grow each time we call push_back and the memory will be re-allocated. This results in a copy of the old data into the newly allocated array. If you are sure you will only be adding a few items then this is most likely not an issue. However in cases where you are likely to need to add hundreds or even thousands of items, the time required to reallocate memory and copy data in this manner will quickly add up. The solution to this problem is to pre-allocate a suitably sized amount of memory so that you are free to add items without having to reallocate as often.

The vector exposes two functions which allow us to do this. The first is vector::resize. This function will resize the vector to the size specified in its first parameter. This number represents the total number of items the vector should be capable of storing before it needs to grow. If we were to pass the value 1000, the vector would be capable of storing 1000 separate int variables in our above example. The second parameter is a value used to initialize the new entries.

```
// Add 1000 new integer items to the vector
m_IntVector.resize( 1000 );

// Add 1000 more, these should be initialized for us
// Notice that the function expects the absolute / total size
m_IntVector.resize( 2000, 5 );
```

Pre-allocating a vector in this manner has certain drawbacks. First, we have only set the overall size of the vector. If we were to call push_back on a vector that had been resized to 1000 the vector would still grow and we would have 1001 items stored. The second drawback is that we would have to maintain a separate variable to keep track of how many variables we have placed into the vector so far. This way when we next wanted to add an item, we could assign it to an existing vector element rather than to the end.

What we would rather do then is just reserve memory rather than resize the vector. vector::reserve allows you to reserve however much memory you need and you can continue to call push_back until it reaches the reserved size. At that point the vector would begin to grow again unless you reserved more memory.

```
// Reserve 1000 elements for us to use
m_IntVector.reserve( 1000 );

// Lets be nasty and add 1000 items
for ( i = 0; i < 1000; i++ ) m_IntVector.push_back( 3 );

// I want to add 1000 new items now, so we need some more room
// Notice that the function expects the absolute / total size
m_IntVector.reserve( 2000 );

// Add some more items.
for ( i = 0; i < 1000; i++ ) m_IntVector.push_back( 3 );
```

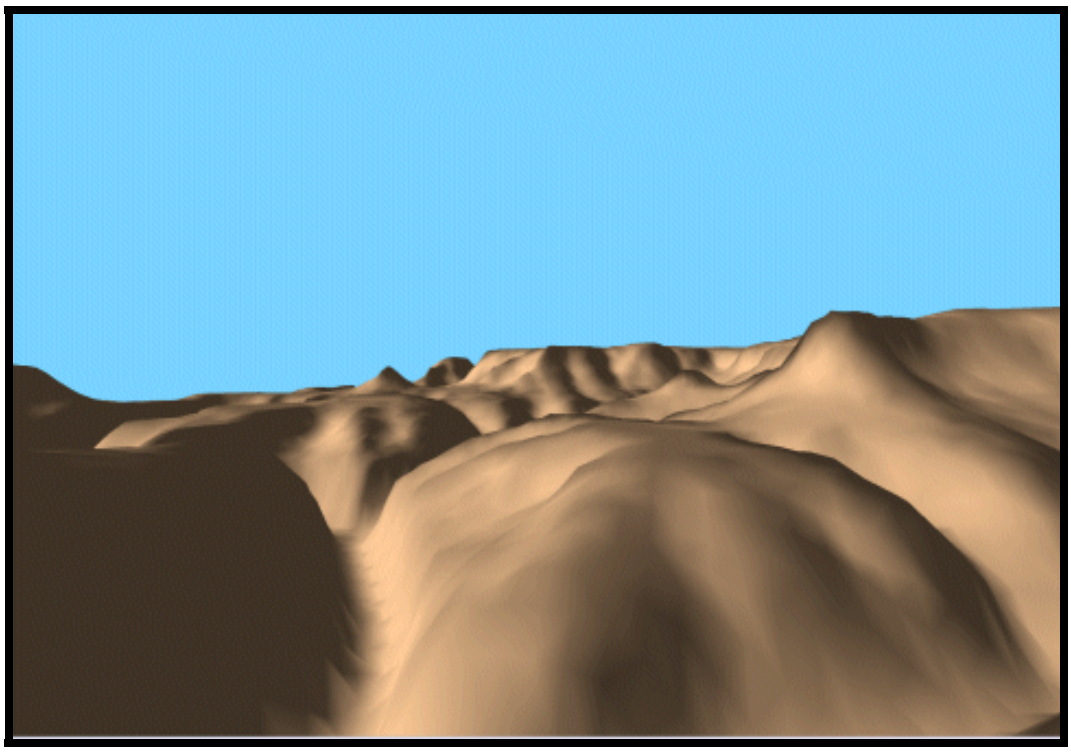
vector::capacity() returns the number of items that are currently reserved in the vector.

Setting and Retrieving Vector Elements.

The STL vector can be accessed much like a standard array since the vector class overloads the [ ] operator:

```
int IntVar = 5;

// Add an element to the vector
m_IntVector.push_back(  IntVar );

// Lets read it back out just for fun
IntVar = m_IntVector[ 0 ];

// Finally we'll adjust the value and assign it again :)
m_IntVector[0] = IntVar;
```

Workbook Chapter Three: Vertex and Index Buffers

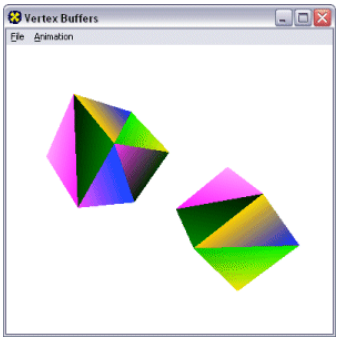


The Lab Projects in this lesson will teach us how to:

- create and use vertex buffers for primitive rendering
- load height map image data
- generate terrain geometry
- create and use index buffers for indexed primitive rendering
- represent complex meshes using indexed triangle strips
- use dynamic vertex buffers for simple animation tasks

Note: Lab Project 3.2 contains source code for a camera class that you can use in your own applications. It implements multiple camera modes such as a First Person, Third Person, and Cockpit. The code will be discussed in detail in Chapter 4.

Lab Project 3.1: Primitive Rendering with Vertex Buffers



Fortunately, this will be our last cube demo for a while. The code is essentially the same as the code we used in Chapter 2 except we will use vertex buffers for rendering. We will briefly discuss only the relevant code changes.

The first change to the application is that we have disposed of the CMesh and CPolygon classes. Recall that until now each object (CObject) had an array of CPolygon objects. Each CPolygon contained an array of CVertex objects. Now we have only a CObject class and a CVertex class. The CVertex class is unchanged from Chapter 2; it holds a position and a diffuse color. Each CObject now includes a pointer to the IDirect3DVertexBuffer9 interface pointing to a vertex buffer that contains a cube mesh. The vertex buffer is essentially a replacement for the CMesh class. We will now be able to render the entire cube (all 12 triangles) with a single call to DrawPrimitive as a triangle list. This is in contrast to our last demo where each face (two triangles) was rendered using separate calls to DrawPrimitiveUP.

> **Note:** Always aim to reduce calls to DrawPrimitive. With a real game level where polygon counts are higher than our simple two-cube example, your application should be trying to render 200 – 500 triangles per call.

The CObject Class

The new CObject class can be found in CObject.h and CObject.cpp. Here is the new class definition:

```
class CObject
{
  public:
  //--------------------------------------------------------------------------
  // Constructors & Destructors for This Class.
  //--------------------------------------------------------------------------

    CObject( LPDIRECT3DVERTEXBUFFER9 pVertexBuffer );
    CObject( );
    virtual ~CObject( );
    void SetVertexBuffer ( LPDIRECT3DVERTEXBUFFER9 pVertexBuffer );
 //--------------------------------------------------------------------------
 // Public Variables for This Class
 //--------------------------------------------------------------------------
    D3DXMATRIX              m_mtxWorld;         // Objects world matrix
    LPDIRECT3DVERTEXBUFFER9 m_pVertexBuffer;    // Vertex Buffer we are instancing
};
```

The class holds only two pieces of information: a world matrix and a pointer to a vertex buffer.

The default constructor performs simple initialization:

```
CObject::CObject()
{
    // Reset / Clear all required values
    m_pVertexBuffer = NULL;
    D3DXMatrixIdentity( &m_mtxWorld );
}
```

The second constructor accepts a pointer to an IDirect3DVertexBuffer9 interface and copies it into its own internal variables. Notice that it increases the reference count of the interface. Be sure you get used to reference count management as it will save you a headache later on when you cannot figure out why an object is being destroyed early (or not being destroyed at all -- causing a memory leak).

```
CObject::CObject( LPDIRECT3DVERTEXBUFFER9 pVertexBuffer )
{
    // Reset / Clear all required values
    D3DXMatrixIdentity( &m_mtxWorld );

    // Set Vertex Buffer
    m_pVertexBuffer = pVertexBuffer;
    m_pVertexBuffer->AddRef();
}
```

We include a SetVertexBuffer function that can be used to assign the vertex buffer to the object:

```
CObject::SetVertexBuffer ( LPDIRECT3DVERTEXBUFFER9 pVertexBuffer )
{
    // Make sure to release any previous interface
    if (m_pVertexBuffer) m_pVertexBuffer->Release();

    // Set Vertex Buffer
    m_pVertexBuffer = pVertexBuffer;

    // If we are setting it to null then bail
    if (!m_pVertexBuffer) return;
    m_pVertexBuffer->AddRef();
}
```

The destructor releases the object's claim on the interface by decreasing the reference count and assigning its own vertex buffer interface pointer to NULL.

```
CObject::~CObject( )
{
    // Release our vertex buffer (de-reference)
    if ( m_pVertexBuffer ) m_pVertexBuffer->Release();
    m_pVertexBuffer = NULL;
}
```

The CGameApp Class

CGameApp::BuildObjects

The CGameApp::BuildObjects function is where our cube mesh is constructed. Unlike the last demo, a static vertex buffer is created and the vertices will be stored there. The CGameApp class maintains primary ownership of the vertex buffer that will be used by both of the game objects in our world.

```
bool CGameApp::BuildObjects()
{
    HRESULT  hRet;
    CVertex *pVertex = NULL;
    ULONG    ulUsage = D3DUSAGE_WRITEONLY;

    // Seed the random number generator
    srand( timeGetTime() );

    // Release previously built objects
    ReleaseObjects();

    // Build our buffers usage flags (i.e. Software T&L etc)
    VERTEXPROCESSING_TYPE vp = m_D3DSettings.GetSettings()->VertexProcessingType;
    if ( vp != HARDWARE_VP && vp != PURE_HARDWARE_VP )
       ulUsage |= D3DUSAGE_SOFTWAREPROCESSING;

    // Create our vertex buffer ( 36 vertices (6 verts * 6 faces) )
    hRet = m_pD3DDevice->CreateVertexBuffer( sizeof(CVertex) * 36, ulUsage,
                                             D3DFVF_XYZ | D3DFVF_DIFFUSE,
```

```
                                        D3DPOOL_MANAGED, &m_pVertexBuffer, NULL );
    if ( FAILED( hRet ) ) return false;
```

The first step is to determine whether or not we want to create the vertex buffer with the D3DUSAGE_SOFTWAREPROCESSING flag. If the device selected after enumeration is a hardware device then we cannot specify this flag. If we are using a software device then we should specify this flag (even though the behavior is implied). The CGameApp class stores a copy of the CD3DSetting structure used to create the device so we can query this structure for the current vertex processing type being used by the device. In this example, the flag is only used if we have a software vertex-processing device, or if we are using a mixed mode device that has currently been set by the user to use software vertex processing.

We call IDirect3DDevice9::CreateVertexBuffer to create a managed static vertex buffer. We expect maximum performance due to the D3DUSAGE_WRITEONLY flag being specified. On a hardware vertex-processing device, the vertex buffer will be created in video memory and a system memory copy will be maintained by the device object. The vertex buffer will automatically be restored should the device become lost and then reset.

If the vertex buffer was created successfully, we will store the interface pointer returned in CGameApp::m_pVertexBuffer.

Because we are using a triangle list, we will have duplicated position vertices. We will need 36 vertices to represent the cube (six faces w/ two triangles each -- 6*2*3 = 36). We will add the vertices to the buffer using a Lock call with a local CVertex pointer. We will use that pointer to iterate through the vertex buffer and add data.

```
    // Lock the vertex buffer ready to fill data
    hRet = m_pVertexBuffer->Lock( 0, sizeof(CVertex) * 36, (void**)&pVertex, 0 );

    if ( FAILED( hRet ) ) return false;

    // Front Face
    *pVertex++ = CVertex( -2,  2, -2, RANDOM_COLOR );
    *pVertex++ = CVertex(  2,  2, -2, RANDOM_COLOR );
    *pVertex++ = CVertex(  2, -2, -2, RANDOM_COLOR );

    *pVertex++ = CVertex( -2,  2, -2, RANDOM_COLOR );
    *pVertex++ = CVertex(  2, -2, -2, RANDOM_COLOR );
    *pVertex++ = CVertex( -2, -2, -2, RANDOM_COLOR );

    // Top Face
    *pVertex++ = CVertex( -2,  2,  2, RANDOM_COLOR );
    *pVertex++ = CVertex(  2,  2,  2, RANDOM_COLOR );
    *pVertex++ = CVertex(  2,  2, -2, RANDOM_COLOR );

    *pVertex++ = CVertex( -2,  2,  2, RANDOM_COLOR );
    *pVertex++ = CVertex(  2,  2, -2, RANDOM_COLOR );
    *pVertex++ = CVertex( -2,  2, -2, RANDOM_COLOR );

    // Back Face
    *pVertex++ = CVertex( -2, -2,  2, RANDOM_COLOR );
```

```
    *pVertex++ = CVertex(  2, -2,  2, RANDOM_COLOR );
    *pVertex++ = CVertex(  2,  2,  2, RANDOM_COLOR );

    *pVertex++ = CVertex( -2, -2,  2, RANDOM_COLOR );
    *pVertex++ = CVertex(  2,  2,  2, RANDOM_COLOR );
    *pVertex++ = CVertex( -2,  2,  2, RANDOM_COLOR );

    // Bottom Face
    *pVertex++ = CVertex( -2, -2, -2, RANDOM_COLOR );
    *pVertex++ = CVertex(  2, -2, -2, RANDOM_COLOR );
    *pVertex++ = CVertex(  2, -2,  2, RANDOM_COLOR );

    *pVertex++ = CVertex( -2, -2, -2, RANDOM_COLOR );
    *pVertex++ = CVertex(  2, -2,  2, RANDOM_COLOR );
    *pVertex++ = CVertex( -2, -2,  2, RANDOM_COLOR );

    // Left Face
    *pVertex++ = CVertex( -2,  2,  2, RANDOM_COLOR );
    *pVertex++ = CVertex( -2,  2, -2, RANDOM_COLOR );
    *pVertex++ = CVertex( -2, -2, -2, RANDOM_COLOR );

    *pVertex++ = CVertex( -2,  2,  2, RANDOM_COLOR );
    *pVertex++ = CVertex( -2, -2, -2, RANDOM_COLOR );
    *pVertex++ = CVertex( -2, -2,  2, RANDOM_COLOR );

    // Right Face
    *pVertex++ = CVertex(  2,  2, -2, RANDOM_COLOR );
    *pVertex++ = CVertex(  2,  2,  2, RANDOM_COLOR );
    *pVertex++ = CVertex(  2, -2,  2, RANDOM_COLOR );

    *pVertex++ = CVertex(  2,  2, -2, RANDOM_COLOR );
    *pVertex++ = CVertex(  2, -2,  2, RANDOM_COLOR );
    *pVertex++ = CVertex(  2, -2, -2, RANDOM_COLOR );

    // Unlock the buffer
    m_pVertexBuffer->Unlock( );
```

We now have a vertex buffer that contains the vertices for the cube mesh. Next we set each object's internal vertex buffer interface pointer so that it uses this vertex buffer object.

```
    // Our two objects should reference this vertex buffer
    m_pObject[ 0 ].SetVertexBuffer ( m_pVertexBuffer );
    m_pObject[ 1 ].SetVertexBuffer ( m_pVertexBuffer );
```

The last part of the function generates an initial world matrix for each of the two cube objects.

```
    // Set both objects matrices so that they are offset slightly
    D3DXMatrixTranslation( &m_pObject[ 0 ].m_mtxWorld, -3.5f,  2.0f, 14.0f );
    D3DXMatrixTranslation( &m_pObject[ 1 ].m_mtxWorld,  3.5f, -2.0f, 14.0f );

    // Success!
    return true;
}
```

CGameApp::SetupRenderStates

CGameApp::SetupRenderStates is called before the main rendering loop begins. We have added a call to the IDirect3DDevice9::SetStreamSource function to bind the vertex buffer of the first object to stream 0. Both objects in our game world have pointers to the same vertex buffer -- much like the shared CMesh in our previous projects. We set the stream source here because we will not have to change vertex buffers during this application.

```
void CGameApp::SetupRenderStates()
{
    // Set up new perspective projection matrix
    float fAspect = (float)m_nViewWidth / (float)m_nViewHeight;
    D3DXMatrixPerspectiveFovLH(&m_mtxProjection, D3DXToRadian(60.0f),
                               fAspect, 1.01f, 1000.0f);

    // Setup our D3D Device initial states
    m_pD3DDevice->SetRenderState( D3DRS_ZENABLE, D3DZB_TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_DITHERENABLE,  TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_SHADEMODE, D3DSHADE_GOURAUD );
    m_pD3DDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CCW );
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, FALSE );

    // Setup our vertex FVF code
    m_pD3DDevice->SetFVF( D3DFVF_XYZ | D3DFVF_DIFFUSE );
    m_pD3DDevice->SetStreamSource( 0, m_pObject[i].m_pVertexBuffer, 0, sizeof(CVertex) );

    // Setup our matrices
    m_pD3DDevice->SetTransform( D3DTS_VIEW, &m_mtxView );
    m_pD3DDevice->SetTransform( D3DTS_PROJECTION, &m_mtxProjection );
}
```

CGameApp::FrameAdvance

The FrameAdvance function has been simplified now that we are using vertex buffers. Most of the code is unchanged from the previous chapter so we will look only at the scene rendering portions that have changed:

```
    // Animate the two objects
    AnimateObjects();

    // Clear the frame & depth buffer ready for drawing
    m_pD3DDevice->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, 0xFFFFFFFF, 1.0f, 0);

    // Begin Scene Rendering
    m_pD3DDevice->BeginScene();

    // Loop through each object
    for ( ULONG i = 0; i < 2; i++ )
    {
        // Set our object matrix
        m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_pObject[i].m_mtxWorld );
```

```
        // Set the vertex stream source
        m_pD3DDevice->SetStreamSource( 0, m_pObject[i].m_pVertexBuffer,
                                       0, sizeof(CVertex) );

        // Render the primitive (Hardcoded, 12 primitives)
        m_pD3DDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 12 );

    } // Next Object

    // End Scene Rendering
    m_pD3DDevice->EndScene();

    // Present the buffer
    if ( FAILED(m_pD3DDevice->Present( NULL, NULL, NULL, NULL )) ) m_bLostDevice = true;
```

Notice that we rendered the cube mesh with one call to DrawPrimitive. Rendering polygons in small batches (1 polygon at a time for example) on a hardware vertex-processing device typically results in only 1% – 5% of the potential power of the GPU being used. Vertex buffers encourage us to batch primitives and this is required for optimal performance.

That is all there is to our first vertex buffer application. Please make sure that you fully understand the code in this project before continuing. From this point on in the course we will use vertex buffers for all primitive rendering. They are faster for the GPU to process but they must be created with care so that we get the best performance we can out of them. Unfortunately, it is often easier to create a vertex buffer the wrong way than the right way.

Lab Project 3.2: Basic Terrain Demo

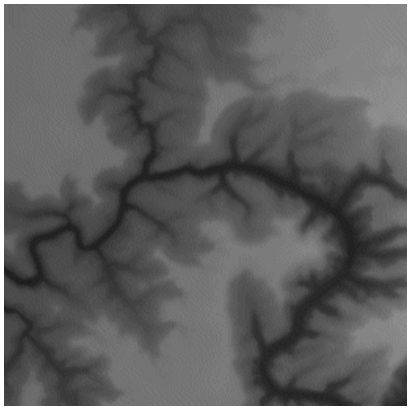
In this project we will take a step beyond our low polygon cubes and create a series of meshes totaling 66,049 vertices in all. Do not fear, we are not going to code all of these vertices by hand. Instead we will be using a 2D image to help generate the mesh. This image will be 257x257 pixels and each pixel in the image will be a shade of gray. If we use only one color component of each pixel (ex. the red component) then this value will be a number between 0 and 255. We will take this color value and multiply it by a scaling factor to assign a height to each vertex in the mesh. The image used for this purpose is referred to as a **height map** because each pixel in the image represents a vertex (or to be more precise, a vertex height value).

Height Maps

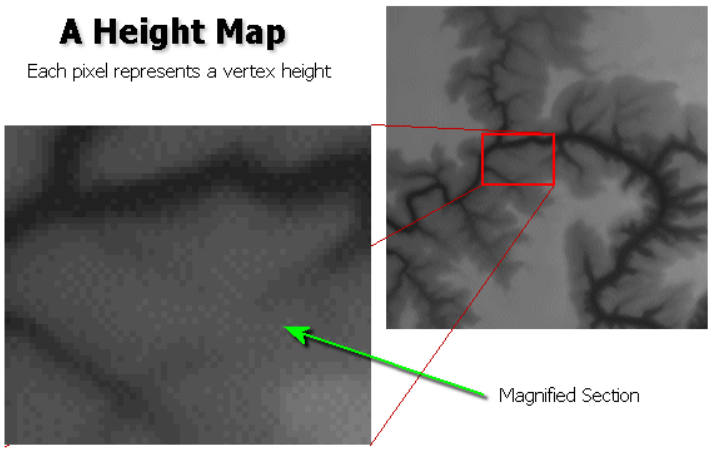
Lab Project 3.2 includes a folder called '*Data*' which contains the image that will be used for the height map. The image is in .RAW format. This format is supported by most popular paint and image editing packages. It is a grayscale image, using 8 bits per pixel. In this demo we will use the value of each pixel to generate a world space pixel height for a vertex.

RAW files are very easy to read since they have no header or any other extraneous information. They simply contain a sequential list of color values. Our height map will be 257x257 in dimensions. Because each pixel represents a single vertex, our terrain will be 257x257 vertices in its dimensions as well (total = 66,049 vertices).

If you would like to view the image in a package such as Paint Shop Pro, simply open it up as a RAW file and fill in its dimensions as 257x257. This is important to do because the file contains no header and the program would have no idea what dimensions the file should be. The height map used in our demo is seen below:



All we will need to do is load this image into a height array (a height map). Technically, we will only need to load the red component of each pixel into the height map. Once we have the array, we can use each pixel position and color to generate a vertex position in world space.



For example, the first pixel in the height map is at position **x=0 : y=0**. The x coordinate of this pixel will be used as the x component of the first world space vertex and the y component will be used as its z component. The height value extracted from the pixel color's red component will be used as the Y component of the first world space vertex.

The following table shows how pixel positions and color values in the image could be used to create world space vertex positions.

Image Pixel Position	Pixel Color (Grayscale)	Generated World Space Vertex
X=5 : Y=15	100 = RGB (**100** , 100 , 100)	X:5 Y:100 Z:15
X=0 : Y=100	64 = RGB (**64** , 64 , 64)	X:0 Y:64 Z:100
X=134 : Y=200	127 = RGB (**127** , 127 , 127)	X:134 Y:127 Z:200

Normally we will want to scale the image pixel positions by some amount or else the height of the vertices will be limited to the 0-255 range and all of the vertices will be located only 1 world space unit away from their neighbouring vertices along the X and Z axes. A larger scale is preferred in most cases. For example, we might decide to scale the image space pixel positions by a factor of 2 so that the terrain is twice as large in the X and Z dimensions. We may also decide to scale the height values by 4 to provide the topology more definition

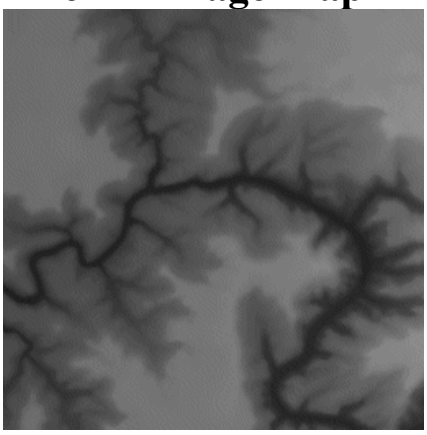
D3DXVECTOR3 ScaleVector (2.0f, 4.0f, 2.0);

The following table shows the same image pixel positions and colors with the world space vertex positions after they have been multiplied by the scale vector.

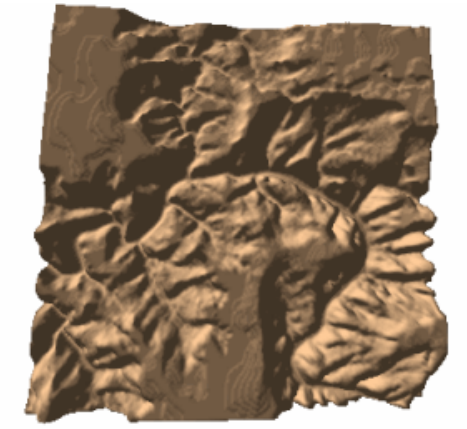
Image Pixel Position	Pixel Color (GreyScale)	Generated World Space Vertex
X=5 : Y=15	100 = RGB (**100** , 100 , 100)	X:10 Y:400 Z:30
X=0 : Y=100	64 = RGB (**64** , 64 , 64)	X:0 Y:256 Z:200
X=134 : Y=200	127 = RGB (**127** , 127 , 127)	X:268 Y:508 Z:400

The following images show the results of using this very straightforward technique. Notice that the lighter the pixel is in the image, the higher the vertex generated from it will be. Look at how the dark strips on the image map create deep ravines in the terrain:

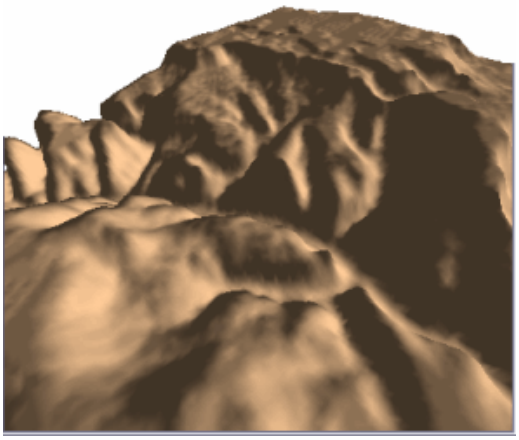
The 2D Image Map



The 3D Terrain



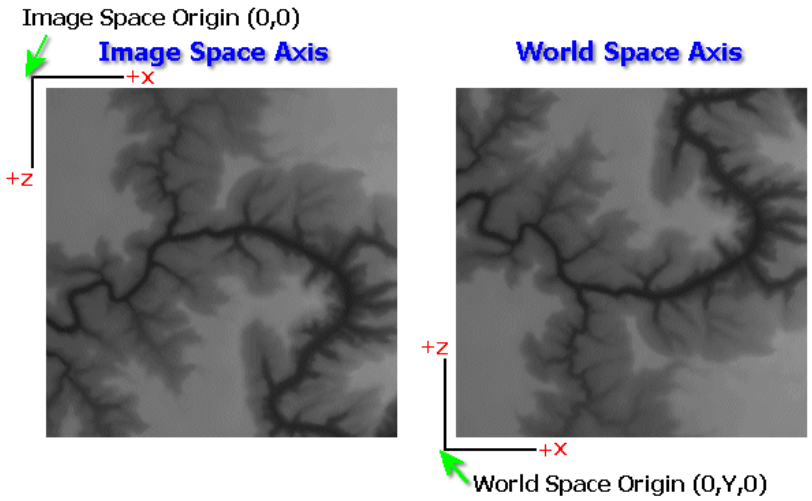
When the camera is placed onto the terrain you can see that even an un-textured terrain can look quite impressive (certainly better than two cubes).



The first thing to consider before we start looking at code is the fact that the image space Y coordinate is being used (indirectly) as the world space Z coordinate. If this is not immediately clear to you then imagine that you have moved the camera so that it is far above the terrain looking down on the center

such that the 3D terrain just about fills the entire screen. Also imagine that the camera orientation is such that your look vector is aligned with the world's negative Y-axis (straight down). This means that you are perpendicular to the terrain such that it looks like a 2D image. You might think that viewing the terrain from this perspective would produce an image of the terrain similar to the height map image. But instead what you will see is that the terrain appears flipped on its horizontal axis. The world space coordinate (x=0, y=0) would actually be at the bottom left of the frame buffer and not the top left as in image space. This is because given the way we are looking at the terrain, the world Z-axis increases going up the screen whilst in image space the Y-axis (which is mapped to the world space Z-axis) increases going down the screen.

The left image below shows the height map with the image space axis origin in its top left corner. The image on the right shows how the terrain would look in world space. The X-axis in both coordinate systems moves in a positive direction from left to right. The Y-axis in image space (which becomes the Z-axis in world space) gets inverted.



As you can see in the above images, the top left corner of the image will be at world space position (0, 0). Of course, we could choose to move the entire terrain to any position in the 3D world simply by translating the vertices by some arbitrary amount. You could also choose to change your indexing strategy when reading the values out of the height map during terrain construction to avoid the inversion effect, but we have decided to keep things simple for this demonstration.

Buffer Size and Primitive Batching

It might seem logical to build one big terrain mesh vertex buffer (and index buffer), and render the entire terrain with a single call to DrawIndexedPrimitive. After all, we have already discussed that we want to minimize calls to DrawIndexedPrimitive. However, while it is true that batching is vitally important, there are limits to this strategy. Sending too much data to the pipeline can actually cause performance to begin to drop off. Hardware manufacturers like nVidia® recommend keeping the number of primitives rendered in a single call to DrawPrimitive or DrawIndexedPrimitive to 200+

triangles. (*Note that 200 triangles rendered as a non-indexed triangle list translates to 600 vertices*.) While we should always try to send at least this many triangles to the card when possible, the + part of this 200+ concept could use a little more examination. So to get a better feel for the range, we decided to break our terrain into different sized meshes and observe the effect on frame rate. These results are listed below. In all cases, the same total number of triangles was rendered. The tests subdivided the terrain mesh into separate meshes where each sub mesh required its own DrawIndexedPrimitive call.

Terrain SubMeshes	SubMesh Size	Number Of Triangles	Frames Per Second	Millions of Triangles per Second
8 x 8	33 x 33	2141	181	24,801,344
4 x 8	64 x 33	4189	177	23,726,496
4 x 4	64 x 64	8381	176	23,600,896
2 x 4	129 x 65	16573	174	23,069,616
2 x 2	129 x 129	33149	173	22,939,108
8 x 16	33 x 17	1069	179	24,492,928
16 x 16	17 x 17	557	191	27,235,072
32 x 32	9 x 9	149	79	12,053,504

The results are interesting. The best performance came when we broke the terrain down into a grid of 16x16 separate sub meshes, each containing 17x17 vertices in its vertex buffer. This worked out to 557 triangles being rendered with each call to DrawIndexedPrimitive until the entire terrain was rendered.

We also see that the 16x16 case outperformed the terrain that was broken into 2x2 large sub meshes. In that case, each index buffer contained 33,149 triangles and required only four calls to the DrawIndexPrimitive function. If sending as many vertices as possible was the overriding factor, then this approach should have produced the best results. But on our test machine it actually turned out to be next to the bottom performance wise. The lowest score came when we subdivided the mesh too much and stored only 149 triangles in each sub mesh. The result was many more calls to DrawIndexedPrimitive. While 79 frames per second may not seem like a bad score, note that performance dropped by almost 60% (112 fps) compared to the best case.

So we can see that sending too many triangles or too few triangles affects performance; the latter seeming to be less preferable. Based on the results of these tests (not conclusive across all hardware by any means) we have decided that for this project we will create our terrain as a grid of 16x16 sub meshes. To be clear, this means we will have 256 separate meshes each with their own vertex buffer. Rendering the terrain will consist of looping through each sub mesh, setting its vertex and index buffers, and calling DrawIndexedPrimitive.

You are encouraged to use similar testing strategies to benchmark any application you write so that you can be at least fairly confident that you are using the appropriately sized data structures (vertex buffers and index buffers in this case) for best performance. This is not an exact science. A good size on one hardware configuration might not produce the same results on another but you can begin to

develop some good approximations. At the very least, you will begin to develop the habit of benchmarking your code and testing your assumptions.

Application Framework Changes

The changes to the CGameApp class are fairly straightforward. We are going to move the rendering and management of the terrain into its own class (see CTerrain.h and CTerrain.cpp). We have also added a CCamera class for view and projection matrix maintenance. The camera class implements three different camera styles: 1st person, 3rd person, and spacecraft. The CCamera class will be explained in detail in Chapter 4.

We will use the same untransformed and pre-lit vertex format as in previous demos. We will not use the CObject class in this demonstration since we will not need to instance meshes. The main change is in the CMesh class since it now includes a vertex buffer and an index buffer with helper functions to aid in their management. Each CMesh object will represent one of the sub meshes of our terrain (there will be 256 of these) and each one will store 17x17 vertices. The CTerrain class will manage an array of these CMesh objects and will be responsible for rendering them when needed.

The CMesh Class

Let us take a look first at the new CMesh class (see CObject.h and CObject.cpp).

```
class CMesh
{
public:
   // Constructors & Destructors for This Class.
   CMesh( ULONG VertexCount, ULONG IndexCount );
   CMesh();
   virtual ~CMesh();

   //  Public Functions for This Class
   long            AddVertex( ULONG Count = 1 );
   long            AddIndex ( ULONG Count = 1 );

   HRESULT BuildBuffers(LPDIRECT3DDEVICE9 pD3DDevice, bool HardwareTnL,
                        bool ReleaseOriginals = true );

   // Public Variables for This Class
   ULONG                    m_nVertexCount;     // Number of vertices stored
   CVertex                 *m_pVertex;           // Simple temporary vertex array
   ULONG                    m_nIndexCount;      // Number of indices stored
   USHORT                  *m_pIndex;           // Simple temporary index array
   LPDIRECT3DVERTEXBUFFER9 m_pVertexBuffer;    // Vertex Buffer to be Rendered
   LPDIRECT3DINDEXBUFFER9  m_pIndexBuffer;     // Index Buffer to be Rendered
};
```

CVertex *m_pVertex
ULONG m_nVertexCount
m_pVertex is a pointer to a temporary vertex array (m_nVertexCount defines the number of vertices in the array). It is used to hold vertices that were added via CMesh::AddVertex until such a time as the vertex buffer is created using CMesh::BuildBuffers. This was done simply for convenience. This memory is freed when the vertex buffer is created, although there may be situations when you might prefer to keep the local memory copy around (to rebuild the buffers after a lost device is recovered if you are using the default pool for example).

USHORT *m_pIndex
ULONG m_nIndexCount
Just as the vertices added to the class are contained in a temporary array until the vertex buffer is created and ready to be filled, so too are the indices. The application calls the CMesh::AddIndex function to add another index to the temporary array. Once all the indices have been added, the application will call BuildBuffers. This will create the vertex and index buffer and will copy the indices and vertices from the temporary arrays into the vertex and index buffers and release the memory that was being used by the temporary arrays.

LPDIRECT3DVERTEXBUFFER9 m_pVertexBuffer
LPDIRECT3DINDEXBUFFER9 m_pIndexBuffer
Vertex and index buffer interface pointers that are created by the BuildBuffers function.

CMesh::CMesh ()

The default constructor sets all pointers to NULL and initializes all variables to zero:

```
CMesh::CMesh()
{
    // Reset / Clear all required values
    m_pVertex       = NULL;
    m_pIndex        = NULL;
    m_nVertexCount  = 0;
    m_nIndexCount   = 0;

    m_pVertexBuffer = NULL;
    m_pIndexBuffer  = NULL;
}
```

The next constructor allows you to specify how many vertices and indices you will need so that it can pre-allocate the temporary arrays. This avoids later resizing when one vertex is added at a time and should reduce fragmentation.

```
CMesh::CMesh( ULONG VertexCount, ULONG IndexCount )
{
    // Reset / Clear all required values
    m_pVertex       = NULL;
    m_pIndex        = NULL;
    m_nVertexCount  = 0;
```

```
    m_nIndexCount   = 0;

    m_pVertexBuffer = NULL;
    m_pIndexBuffer  = NULL;

    // Add Vertices & indices if required
    if ( VertexCount > 0 ) AddVertex( VertexCount );
    if ( IndexCount  > 0 ) AddIndex( IndexCount );
}
```

CMesh::~CMesh ()

The destructor releases the vertex buffer and index buffers and deletes the temporary arrays (if they have not already been deleted by the BuildBuffers function).

```
CMesh::~CMesh()
{
    // Release our mesh components
    if ( m_pVertex ) delete []m_pVertex;
    if ( m_pIndex  ) delete []m_pIndex;

    if ( m_pVertexBuffer ) m_pVertexBuffer->Release();
    if ( m_pIndexBuffer  ) m_pIndexBuffer->Release();

    // Clear variables
    m_pVertex       = NULL;
    m_pIndex        = NULL;
    m_nVertexCount  = 0;
    m_nIndexCount   = 0;

    m_pVertexBuffer = NULL;
    m_pIndexBuffer  = NULL;
}
```

CMesh::AddVertex

The AddVertex function allows us to add more space to our temporary vertex array for additional vertices. To do this it has to create a new array large enough to hold the old vertices and the new amount to be added. The old vertices are copied and the previous temporary array is released.

```
long CMesh::AddVertex( ULONG Count )
{
    CVertex * pVertexBuffer = NULL;

    // Allocate new resized array
    if (!( pVertexBuffer = new CVertex[ m_nVertexCount + Count ] )) return -1;

    // Existing Data?
    if ( m_pVertex )
    {
        // Copy old data into new buffer
```

```
        memcpy( pVertexBuffer, m_pVertex, m_nVertexCount * sizeof(CVertex) );

        // Release old buffer
        delete []m_pVertex;

    } // End if

    // Store pointer for new buffer
    m_pVertex = pVertexBuffer;
    m_nVertexCount += Count;

    // Return first vertex
    return m_nVertexCount - Count;
}
```

The AddIndex function is the same as above with the exception that it resizes the temporary index array.

CMesh::BuildBuffers

BuildBuffers builds the vertex and index buffers from the two temporary arrays. We pass it a pointer to the IDirect3DDevice9 interface, a Boolean specifying whether we are creating this mesh for a hardware or software vertex-processing device, and a boolean specifying whether we want the temporary arrays to be freed after the vertex and index buffers have been created. This boolean is here because you may want to keep the temporary arrays around if you need to rebuild the vertex buffers at a later date (eg. if the device is lost while using the **D3DPOOL_DEFAULT** pool).

```
HRESULT CMesh::BuildBuffers( LPDIRECT3DDEVICE9 pD3DDevice, bool HardwareTnL, bool
ReleaseOriginals )
{
    HRESULT     hRet     = S_OK;
    CVertex    *pVertex  = NULL;
    USHORT     *pIndex   = NULL;
    ULONG       ulUsage  = D3DUSAGE_WRITEONLY;

    // Should we use software vertex processing ?
    if ( !HardwareTnL ) ulUsage |= D3DUSAGE_SOFTWAREPROCESSING;

    // Release any previously allocated vertex / index buffers
    if ( m_pVertexBuffer ) m_pVertexBuffer->Release();
    if ( m_pIndexBuffer  ) m_pIndexBuffer->Release();
    m_pVertexBuffer = NULL;
    m_pIndexBuffer  = NULL;
```

The first thing we do is setup the flags that will be used to create our vertex and index buffers. We use the `D3DUSAGE_WRITEONLY` flag for best performance. We also add the `D3DUSAGE_SOFTWAREPROCESSING` flag if the mesh is being used on a software vertex-processing device. If any vertex buffer or index buffer currently exists, we release it first to avoid a memory leak.

The next block of code populates the buffers using the data in the temporary arrays.

```
    // Create our vertex buffer
    pD3DDevice->CreateVertexBuffer( sizeof(CVertex) * m_nVertexCount, ulUsage,
                                    D3DFVF_XYZ | D3DFVF_DIFFUSE,
                                    D3DPOOL_MANAGED, &m_pVertexBuffer, NULL );

    // Lock the vertex buffer ready to fill data
    m_pVertexBuffer->Lock( 0, sizeof(CVertex) * m_nVertexCount, (void**)&pVertex, 0 );

    // Copy over the vertex data
    memcpy( pVertex, m_pVertex, sizeof(CVertex) * m_nVertexCount );

    // We are finished with the vertex buffer
    m_pVertexBuffer->Unlock();

    // Create our index buffer
    pD3DDevice->CreateIndexBuffer( sizeof(USHORT) * m_nIndexCount, ulUsage, D3DFMT_INDEX16,
                                   D3DPOOL_MANAGED, &m_pIndexBuffer, NULL );

    // Lock the index buffer ready to fill data
    m_pIndexBuffer->Lock( 0, sizeof(USHORT) * m_nIndexCount, (void**)&pIndex, 0 );

    // Copy over the index data
    memcpy( pIndex, m_pIndex, sizeof(USHORT) * m_nIndexCount );

    // We are finished with the indexbuffer
    m_pIndexBuffer->Unlock();
```

If the caller requested that the temporary arrays be destroyed, the memory is freed and the counts reset.

```
       // Release old data if requested
    if ( ReleaseOriginals )
    {
        // Release our mesh components
        if ( m_pVertex ) delete []m_pVertex;
        if ( m_pIndex  ) delete []m_pIndex;

        // Clear variables
        m_pVertex       = NULL;
        m_pIndex        = NULL;
        m_nVertexCount  = 0;
        m_nIndexCount   = 0;

    } // End if ReleaseOriginals

    return S_OK;
}
```

The CGameApp Class

Let us next examine the changes to CGameApp.h. The only change is the addition of two new member variables:

```
CTerrain     m_Terrain;          // Simple terrain object (stores data)
CCamera      m_Camera;           // Camera class used to manipulate our player view
```

The CGameApp class owns a single CTerrain object. This terrain object contains functions for loading the image and generating the height map as well as building the 256 meshes using the height map data. It also is responsible for rendering each mesh. The CCamera object allows the player to move around the game world and will be covered in Chapter 4.

One of the biggest changes to the CGameApp class is in the SetupGameState function. It now uses the CCamera class to manage the view and projection matrices. This means we no longer have to create these matrices within the CGameApp class itself. Matrix initialization is handled by the CCamera class based on the parameters we pass into a member function. The CCamera class uses the CPlayer class to position the camera in the world (the CPlayer class has the Camera attached to it). When the player moves, its attached camera is moved automatically.

CGameApp::SetupGameState

```
void CGameApp::SetupGameState()
{
    // Generate an identity matrix
    D3DXMatrixIdentity( &m_mtxIdentity );

    // App is active
    m_bActive = true;

    m_Player.SetCameraMode( CCamera::MODE_FPS );
    m_pCamera = m_Player.GetCamera();

    // Setup our player's default details
    m_Player.SetFriction( 250.0f ); // Per Second
    m_Player.SetGravity( D3DXVECTOR3( 0, -400.0f, 0 ) );
    m_Player.SetMaxVelocityXZ( 125.0f );
    m_Player.SetMaxVelocityY ( 400.0f );
    m_Player.SetCamOffset( D3DXVECTOR3( 0.0f, 10.0f, 0.0f ) );
    m_Player.SetCamLag( 0.0f );

    // Set up the players collision volume info
    VOLUME_INFO Volume;
    Volume.Min = D3DXVECTOR3( -3, -10, -3 );
    Volume.Max = D3DXVECTOR3(  3,  10,  3 );
    m_Player.SetVolumeInfo( Volume );

    // Setup our cameras view details
    m_pCamera->SetFOV( 60.0f );
    m_pCamera->SetViewport(m_nViewX,m_nViewY,m_nViewWidth,m_nViewHeight,1.01f,5000.0f);
```

```
    // Set the camera volume info (matches player volume)
    m_pCamera->SetVolumeInfo( Volume );

    // Add the update callbacks required
    m_Player.AddPlayerCallback( CTerrain::UpdatePlayer, (LPVOID)&m_Terrain );
    m_Player.AddCameraCallback( CTerrain::UpdateCamera, (LPVOID)&m_Terrain );

    // Lets give a small initial rotation and set initial position
    m_Player.SetPosition( D3DXVECTOR3( 430.0f, 400.0f, 330.0f ) );
    m_Player.Rotate( 25, 45, 0 );
}
```

The CCamera class creates a projection matrix with a FOV of 60 degrees and is set to First Person camera mode. We also send it the dimensions of our front buffer and the distance to the near and far planes in the call to CCamera::SetViewport. All of this will be examined in detail in our next lesson.

CGameApp::SetupRenderStates

SetupRenderStates now includes two new function calls into the CCamera class to instruct the camera to set its internally managed view and projection matrices as the current view and projection matrices for the device. The rest of the states are unchanged from our last demo.

```
void CGameApp::SetupRenderStates()
{
    // Setup our D3D Device initial states
    m_pD3DDevice->SetRenderState( D3DRS_ZENABLE, D3DZB_TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_DITHERENABLE,  TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_SHADEMODE, D3DSHADE_GOURAUD );
    m_pD3DDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CCW );
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, FALSE );

    // Setup our vertex FVF code
    m_pD3DDevice->SetFVF( D3DFVF_XYZ | D3DFVF_DIFFUSE );

    // Update our device with our camera details (Required on reset)
    m_Camera.UpdateRenderView( m_pD3DDevice );
    m_Camera.UpdateRenderProj( m_pD3DDevice );
}
```

We pass in a pointer to the device interface so that the camera can update its view and projection state matrices. CCamera member functions (such as SetPosition and Rotate) alter only the internally managed matrices. We call the UpdateRenderView and UpdateRenderProj functions to commit the changes to the device.

CGameApp::BuildObjects

BuildObjects first checks the device settings to determine if we are using a hardware vertex-processing device. The CTerrain class will want to know this so that it can instruct its CMeshes to build the vertex

and index buffers with the correct flags. We also call ReleaseObjects so that if this function has been called when the game objects already exist, their memory will be released so that they can safely be rebuilt. Next we send the CTerrain class a pointer to the device interface and a Boolean indicating whether it is using software or hardware vertex processing. Then we can call CTerrain::LoadHeightMap to load the file 'HeightMap.Raw' and use it to build a height map. This will also build all of the terrain meshes in turn.

Finally, the function creates a single cube mesh that will be used to render the player object (think of it as a placeholder for an animated character mesh) when the camera is in 3rd person mode. The mesh is added to the CPlayer object using the CPlayer::Set3rdPersonMesh function. The CPlayer class will be covered in Chapter 4 when we examine camera systems.

```
bool CGameApp::BuildObjects()
{
    VERTEXPROCESSING_TYPE vp = m_D3DSettings.GetSettings()->VertexProcessingType;
    bool HardwareTnL = true;

    // Are we using HardwareTnL ?
    if ( vp != HARDWARE_VP && vp != PURE_HARDWARE_VP ) HardwareTnL = false;

    // Release previously built objects
    ReleaseObjects();

    // Build our terrain data
    m_Terrain.SetD3DDevice( m_pD3DDevice, HardwareTnL );
    if ( !m_Terrain.LoadHeightMap( _T("Data\\HeightMap.raw"), 257, 257 )) return false;

    // Build a 'player' mesh (this is just a cube currently)
    CVertex * pVertex = NULL;
    srand( timeGetTime() );

    // Add the 8 cube vertices to this mesh
    if ( m_PlayerMesh.AddVertex( 8 ) < 0 ) return false;

    // Add all 4 vertices
    pVertex = &m_PlayerMesh.m_pVertex[0];

    // Add bottom 4 vertices
    *pVertex++ = CVertex( -3, 0, -3, RANDOM_COLOR );
    *pVertex++ = CVertex( -3, 0,  3, RANDOM_COLOR );
    *pVertex++ = CVertex(  3, 0,  3, RANDOM_COLOR );
    *pVertex++ = CVertex(  3, 0, -3, RANDOM_COLOR );

    // Add top 4 vertices
    *pVertex++ = CVertex( -3, 20, -3, RANDOM_COLOR );
    *pVertex++ = CVertex( -3, 20,  3, RANDOM_COLOR );
    *pVertex++ = CVertex(  3, 20,  3, RANDOM_COLOR );
    *pVertex++ = CVertex(  3, 20, -3, RANDOM_COLOR );

    // Add the indices as a strip (with one degenerate) ;)
    if ( m_PlayerMesh.AddIndex( 16 ) < 0 ) return false;
    m_PlayerMesh.m_pIndex[ 0] = 5;
    m_PlayerMesh.m_pIndex[ 1] = 6;
    m_PlayerMesh.m_pIndex[ 2] = 4;
    m_PlayerMesh.m_pIndex[ 3] = 7;
```

```
    m_PlayerMesh.m_pIndex[ 4] = 0;
    m_PlayerMesh.m_pIndex[ 5] = 3;
    m_PlayerMesh.m_pIndex[ 6] = 1;
    m_PlayerMesh.m_pIndex[ 7] = 2;
    m_PlayerMesh.m_pIndex[ 8] = 3; // Degen Index
    m_PlayerMesh.m_pIndex[ 9] = 7;
    m_PlayerMesh.m_pIndex[10] = 2;
    m_PlayerMesh.m_pIndex[11] = 6;
    m_PlayerMesh.m_pIndex[12] = 1;
    m_PlayerMesh.m_pIndex[13] = 5;
    m_PlayerMesh.m_pIndex[14] = 0;
    m_PlayerMesh.m_pIndex[15] = 4;

    // Build the mesh's vertex and index buffers
    if (FAILED(m_PlayerMesh.BuildBuffers( m_pD3DDevice, HardwareTnL, true ))) return false;

    // Our object references this mesh
    m_Object.m_pMesh = &m_PlayerMesh;

    // Link this object to our player
    m_Player.Set3rdPersonObject( &m_Object );

        return true;
}
```

Notice that cube was created as an indexed triangle strip so that it can be rendered with a single call to DrawPrimitive. We will discuss the technique for creating indexed triangle strips when we talk about rendering the terrain. Appendix A at the end of the chapter details how to represent a cube as an indexed triangle strip.

CGameApp::FrameAdvance

The render loop in CGameApp::FrameAdvance has been simplified since the terrain will handle its own mesh rendering. We call the CTerrain::Render function to draw the terrain meshes. If the camera is in 3rd person mode we also call the CPlayer::Render function to draw the placeholder cube avatar.

```
    // Poll & Process input devices
    ProcessInput();

    // Animate the game objects
    AnimateObjects();

    // Clear the frame & depth buffer ready for drawing
    m_pD3DDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, 0x79D3FF, 1.0f, 0 );

    // Begin Scene Rendering
    m_pD3DDevice->BeginScene();

    // Render our terrain objects
    m_Terrain.Render( );

    // Request our player render itself
    m_Player.Render( m_pD3DDevice );
```

```
    // End Scene Rendering
    m_pD3DDevice->EndScene();

    // Present the buffer
    if ( FAILED(m_pD3DDevice->Present( NULL, NULL, NULL, NULL )) ) m_bLostDevice = true;
```

The ProcessInput function has changed quite a bit now that we can move the camera around, but we will leave that discussion until Chapter 4. AnimateObjects does nothing in this demo since the terrain is not animated; it is left as an empty function.

We note that the terrain sub meshes actually have their vertices specified in world space coordinates. This means we no longer need to perform an object space to world space transformation and can set the device world matrix to an identity matrix for terrain rendering.

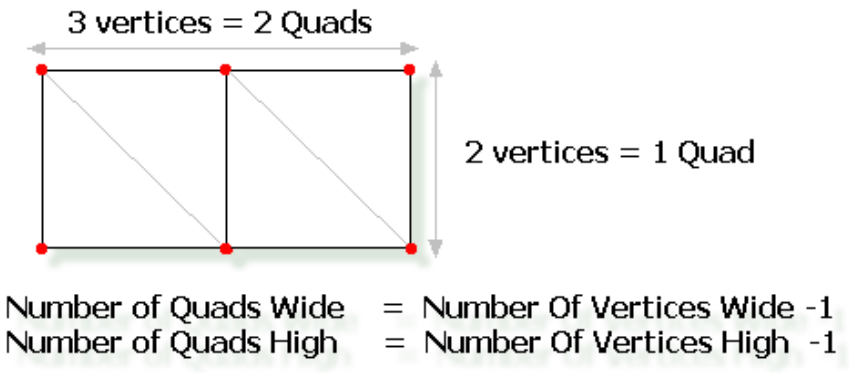
For the IDirect3DDevice9::Clear method we have passed a light blue color for the frame buffer. This provides a simple colored background for our sky.

Between the BeginScene and EndScene calls there is a call to CTerrain::Render for rendering the terrain sub meshes as well as a call for rendering the player mesh (which will only be visible in 3rd person mode).

The CTerrain Class

The CTerrain class will load a height map and construct an array of meshes from that height map such that each mesh represents a portion of the terrain. This means that it has to be able to calculate the vertex positions from the height map and also be able to build an index list so that the vertices form suitable triangles. Rendering the terrain will actually be the easiest part since it only involves looping through each mesh and calling DrawIndexedPrimitive using that mesh's vertex and index buffers. A single call to DrawIndexedPrimitive will render an entire mesh. The terrain for this project will be divided into 256 different meshes. You can change this simply by changing a few variables in the application. Each mesh will be 17 vertices wide and 17 vertices deep. Each group of 4 vertices will be a quad made using two triangles. So each mesh will be 16x16 quads in dimensions because the number of quads is always equal to number of vertices – 1. This is precisely why we used the odd numbers 257x257 for the height map since it creates a terrain of 256x256 quads.

The following image shows how two rows of three vertices (2x3) produce an array of (1x2) quads.



If we used the same diffuse color for every vertex it would be difficult to see any of the terrain detail. This is why lighting is so important in 3D games. Textures improve the situation, but without them it is difficult to tell where each polygon ends and another one begins. Since lighting will not be covered until Chapter 5 we will have to fake some lighting calculations to generate a diffuse color for each vertex. We will color each vertex using an approach that factors in an imaginary light source position in the world and the orientation of the triangles that use the vertex. We will use a brown diffuse color and scale it so that triangles facing away from the light source are cast into shadow.

The CTerrain class definition is found in CTerrain.h:

```
class CTerrain
{
public:
    // Constructors and Destructors
             CTerrain();
    virtual ~CTerrain();

    // Public member functions
    void SetD3DDevice(LPDIRECT3DDEVICE9 pD3DDevice, bool HardwareTnL );
    bool LoadHeightMap(LPCTSTR FileName, ULONG Width, ULONG Height );
    bool LineOfSight(D3DXVECTOR3& vecRayStart, D3DXVECTOR3& vecRayEnd,float Accuracy=0.2f);
    float  GetHeight     ( float x, float z );
    void   Render        ( );
    void   Release       ( );

    // Static call-back functions
    static void     UpdatePlayer  ( LPVOID pContext, CPlayer * pPlayer, float TimeScale );
    static void     UpdateCamera  ( LPVOID pContext, CCamera * pCamera, float TimeScale );


private:
    // private member variables
    D3DXVECTOR3          m_vecScale;         // Amount to scale the terrain meshes
    UCHAR               *m_pHeightMap;       // The physical heightmap data loaded
    ULONG                m_nHeightMapWidth;  // Width of the 2D heightmap data
    ULONG                m_nHeightMapHeight; // Height of the 2D heightmap data

    CMesh               **m_pMesh;           // Simple array of mesh pointers
    ULONG                m_nMeshCount;       // Number of meshes stored here
    LPDIRECT3DDEVICE9    m_pD3DDevice;       // D3D Device to use for creation / rendering.
    bool                 m_bHardwareTnL;     // Used hardware vertex processing ?
```

```
    ULONG               m_nPrimitiveCount;   // Pre-Calculated. Num render primitives
    // private member functions
    long            AddMesh             ( ULONG Count = 1 );
    bool            BuildMeshes         ( );
    D3DXVECTOR3     GetHeightMapNormal  ( ULONG x, ULONG z );
};
```

D3DXVECTOR3 m_vecScale;
Although each pixel in the image describes a vertex position, we will often want to scale the image position to generate a larger terrain. In our demo we will set this vector to (8,2,8). This multiples the image pixel position by 8 so that a pixel at coordinate (2, 4) will generate a vertex at (X=16, Z=32). The red color component of the pixel is multiplied by 2 in this case so the world space vertex heights will be in the range [0, 512].

UCHAR * m_pHeightMap;
This byte array will hold the actual height map data used for building the terrain. As we read the image file, we only read in the first byte of each color (the red component) and store it in this array. Each byte in this array represents the height of a vertex once it is multiplied by the scale vector. The height map and scaling vector are only used to build the terrain meshes. They are not used during rendering, so the height map can be discarded after the meshes have been generated.

ULONG m_nHeightMapWidth;
ULONG m_nHeightMapHeight;
These values store the dimensions of the image file and the height map. We are using a 257x257 image so the height map will be 257x257 as well.

CMesh **m_pMesh;
This is an array of CMesh pointers that will contain pointers to all terrain sub meshes.

ULONG m_nMeshCount;
The number of mesh pointers in the mesh array. This will initially be zero until the terrain is generated. Using our default setting, there should be 256 (16x16) meshes created.

LPDIRECT3DDEVICE9 m_pD3DDevice;
A pointer to the Direct3D device interface.

bool m_bHardwareTnL;
Used to store whether the above device is a software or hardware vertex-processing device.

ULONG m_nPrimitiveCount;
This will be used to store the pre-calculated primitive count for an entire sub mesh. Because we will be rendering a mesh with a single call to DrawIndexedPrimitive, we must tell the device how many primitives to draw. This value has not been hard-coded so that we can easily change the sizes of each mesh to subdivide the terrain to a greater or lesser extent.

CTerrain::CTerrain()

The constructor makes sure our data is initialized.

```
CTerrain::CTerrain()
{
    // Reset all required values
    m_pD3DDevice         = NULL;

    m_pHeightMap         = NULL;
    m_nHeightMapWidth    = 0;
    m_nHeightMapHeight   = 0;

    m_pMesh              = NULL;
    m_nMeshCount         = 0;

    m_vecScale           = D3DXVECTOR3( 1.0f, 1.0f, 1.0f );
}
```

CTerrain::~CTerrain()

The destructor calls the CTerrain::Release() function to clean up memory allocated by the terrain class. Moving the clean up code into its own function lets us release the terrain memory from elsewhere in our application or from elsewhere within the CTerrain class itself when the terrain needs to be rebuilt or simply destroyed.

```
CTerrain::~CTerrain()
{
    Release();
}
```

CTerrain::Release

The Release function deletes the height map and mesh pointer arrays. The CMesh class index and vertex buffers are released in the CMesh destructor. We are also careful to release our claim of usage on the device interface, which will decrement its reference count.

```
void CTerrain::Release()
{
    // Release Heightmap
    if ( m_pHeightMap ) delete[]m_pHeightMap;

    // Release Meshes
    if ( m_pMesh )
    {
        // Delete all individual meshes in the array.
        for ( ULONG i = 0; i < m_nMeshCount; i++ )
        {
```

```
            if ( m_pMesh[i] ) delete m_pMesh[i];

        } // Next Mesh

        // Free up the array itself
        delete []m_pMesh;

    } // End if

    // Release our D3D Object ownership
    if ( m_pD3DDevice ) m_pD3DDevice->Release();

    // Clear Variables
    m_pHeightMap        = NULL;
    m_nHeightMapWidth   = 0;
    m_nHeightMapHeight  = 0;
    m_pMesh             = NULL;
    m_nMeshCount        = 0;
    m_pD3DDevice        = NULL;
}
```

CTerrain::SetD3DDevice

The first CTerrain method to be called from the CGameApp::BuildObjects function is CTerrain::SetD3DDevice. This application passes in a pointer to the device interface used for rendering the terrain. The function simply stores the device interface pointer in its own member variable and increases its reference count. We also pass a boolean (to be stored) indicating whether or not the device passed is a hardware vertex-processing device.

```
void CTerrain::SetD3DDevice( LPDIRECT3DDEVICE9 pD3DDevice, bool HardwareTnL )
{
    // Validate Parameters
    if ( !pD3DDevice ) return;

    // Store D3D Device and add a reference
    m_pD3DDevice = pD3DDevice;
    m_pD3DDevice->AddRef();

    // Store vertex processing type for buffer creation
    m_bHardwareTnL = HardwareTnL;
}
```

CTerrain::LoadHeightMap

CTerrain::LoadHeightMap kick starts the terrain generation process. When this function returns, the terrain meshes will have been constructed and are ready to be rendered. This function takes the RAW file name along with its width and height. We pass these dimensions so that it knows how the data should be organized into rows. A RAW file is essentially a single array of sequential RGB data where each color component is a byte wide. The array is laid out using the following format:

{R,G,B,R,G,B,R,G,B,R,G,B,R,G,B,.....}

```
bool CTerrain::LoadHeightMap( LPCTSTR FileName, ULONG Width, ULONG Height )
{
    FILE *pFile = NULL;

    // Cannot load if already allocated (must be explicitly released for reuse)
    if ( m_pMesh ) return false;

    // Must have an already set D3D Device
    if ( !m_pD3DDevice ) return false;

    // First of all store the information passed
    m_nHeightMapWidth  = Width;
    m_nHeightMapHeight = Height;
```

We return from the function if the CMesh array already exists because these will need to be released first. We also return failure if the class has not yet had its device interface pointer set to a valid IDirect3DDevice9 interface. We store the passed the width and height of the image in member variables.

Next we calculate the scale vector. We found that a scale factor of 4 in the image space X and Y dimensions (world space X and Z) produced a nice sized terrain for a 512x512 height map for our purposes. We use this fact to calculate a good scale value for the X and Z vertex components of any arbitrarily sized height map image so that it scales in proportion to scaling a 512x512 height map by 4.

```
    // Use nice scale
    m_vecScale.x = 4.0f * (512 / (m_nHeightMapWidth - 1));
    m_vecScale.y = 2.0f;
    m_vecScale.z = 4.0f * (512 / (m_nHeightMapHeight - 1));
```

So our height map of 257x257 equates to scale values:

m_vecScale.x = 4.0 * (512 / 256) =8
m_vecScale.z = 4.0 * (512 / 256) =8;

The scale vector Y component can be used to flatten the terrain peaks and ravines (by lowering the value) or to emphasize them (by using a higher value). You should experiment with all of these values yourself to find a combination that works for your application.

Now we allocate memory for the height map array and open the file. We loop through each pixel and read only the first byte of each set of three (R from RGB). The following two bytes are skipped using the fseek function so that we are ready to read the red component of the next adjacent pixel in the file during the next loop iteration.

```
    // Attempt to allocate space for this heightmap information
    m_pHeightMap = new UCHAR[Width * Height];

    // Open up the heightmap file
    pFile = _tfopen( FileName, _T("rb") );
```

```
    // Read the heightmap data (Read only 'Red' component)
    for ( ULONG i = 0; i < Width * Height; i++ )
    {
        fread( &m_pHeightMap[i], 1, 1, pFile );
        fseek( pFile, 2, SEEK_CUR );
    } // Next Value
```

At this point we have read all of the information and it is time to close the file.

```
    // Finish up
    fclose( pFile );
```

The next function that is called is CTerrain::AddMesh. It allocates an array large enough to hold pointers for as many meshes as we need. The next calculation tells the function how many meshes we are going to have to allocate memory for:

```
    // Allocate enough meshes to store the separate blocks of this terrain
    if ( AddMesh( ((Width - 1) / QuadsWide) * ((Height - 1) / QuadsHigh) ) < 0 )
        return false;
```

When rendering a grid-like construct such as a height map, it is more convenient and intuitive to think in terms of quads rather than triangles. The constants QuadsWide and QuadsHigh are defined at the top of the CTerrain.cpp source file. These are wrapped in a nameless namespace so that they cannot be externed to another source code module (deliberately or accidentally).

```
namespace
{
    const USHORT BlockWidth  = 17;            // Number of vertices in a terrain block (X)
    const USHORT BlockHeight = 17;            // Number of vertices in a terrain block (Z)
    const USHORT QuadsWide   = BlockWidth - 1;   // Number of quads in a terrain block (X)
    const USHORT QuadsHigh   = BlockHeight - 1;  // Number of quads in a terrain block (Z)
};
```

This approach allows us to quickly change the subdivision strategy. In this case each mesh will be a grid of 17x17 vertices and thus a 16x16 grid of quads representing a section of the terrain. The AddMesh function is being called as follows:

AddMesh ((256/16) * (256/16)) = AddMesh (16 * 16) = 256 meshes needed

With the mesh array pointer allocated, we now need to build a vertex and index buffer for each mesh using the height map data. This needs to be done such that each mesh can be rendered as a single indexed triangle strip.

```
    // Build the mesh data itself
    return BuildMeshes( );
}
```

Here is the LoadHeightMap function in its entirety (without error checking):

```
bool CTerrain::LoadHeightMap( LPCTSTR FileName, ULONG Width, ULONG Height )
{
    FILE * pFile = NULL;

    if ( m_pMesh ) return false;
    if ( !m_pD3DDevice ) return false;

    // First of all store the information passed
    m_nHeightMapWidth  = Width;
    m_nHeightMapHeight = Height;

    // calculate scale vector
    m_vecScale.x = 4.0f * (512 / (m_nHeightMapWidth - 1));
    m_vecScale.y = 6.0f;
    m_vecScale.z = 4.0f * (512 / (m_nHeightMapHeight - 1));

    // Allocate Heightmap
    m_pHeightMap = new UCHAR[Width * Height];

    pFile = _tfopen( FileName, _T("rb") );

    for ( ULONG i = 0; i < Width * Height; i++ )
    {
        fread( &m_pHeightMap[i], 1, 1, pFile );
        fseek( pFile, 2, SEEK_CUR );
    }

    fclose( pFile );

    // Allocate enough meshes to store the separate blocks of this terrain
    if ( AddMesh( ((Width - 1) / QuadsWide) * ((Height - 1) / QuadsHigh) ) < 0 )
         return false;

    // Build the mesh data itself
    return BuildMeshes( );
}
```

CTerrain::AddMesh

The AddMesh call is used to add new meshes to the CTerrain mesh array. We pass in the number of meshes to make space for. Note that the CTerrain mesh array is an array of CMesh pointers and not CMesh objects.

```
long CTerrain::AddMesh( ULONG Count )
{
    CMesh **pMeshBuffer = NULL;

    // Allocate new resized array
    if (!( pMeshBuffer = new CMesh*[ m_nMeshCount + Count ] )) return -1;

    // Clear out slack pointers
    ZeroMemory( &pMeshBuffer[ m_nMeshCount ], Count * sizeof( CMesh* ) );
```

At this point we have a clean array with enough room to store old and new mesh pointers. If any existed previously, we copy them over into the new array:

```
    if ( m_pMesh )
    {
        // Copy old data into new buffer
        memcpy( pMeshBuffer, m_pMesh, m_nMeshCount * sizeof( CMesh* ) );

        // Release old buffer
        delete []m_pMesh;
    }

    // Store pointer for new buffer
    m_pMesh = pMeshBuffer;
```

Now we can allocate a new CMesh object for each pointer in the array.

```
    // Allocate new mesh pointers
    for ( UINT i = 0; i < Count; i++ )
    {
        // Allocate new mesh
        if (!( m_pMesh[ m_nMeshCount ] = new CMesh() )) return -1;

        // Increase overall mesh count
        m_nMeshCount++;
    }

    // Return first mesh
    return m_nMeshCount - Count;
}
```

When this function returns control to the LoadHeightMap function, the terrain will have an array of 256 CMesh pointers to valid CMesh objects The CMesh objects have not been initialized with any useful data yet. We will do that in the CTerrain::BuildMeshes function which we will examine next.

CTerrain:: BuildMeshes

The first thing BuildMeshes must do is calculate how many blocks wide and how many blocks high the terrain will be. A block in this case actually means a mesh since the entire terrain is basically a rectangular grid of meshes.

```
bool CTerrain::BuildMeshes( )
{
    long x, z, vx, vz, Counter, StartX, StartZ;
    long BlocksWide = (m_nHeightMapWidth  - 1) / QuadsWide;
    long BlocksHigh = (m_nHeightMapHeight - 1) / QuadsHigh;
```

BlocksWide now holds the value of how many meshes the terrain will be divided into along the X-axis. BlocksHigh holds how many meshes will make up the terrain along the Z-axis. We are calculating how many quads in total are required in the X and Z dimensions for the entire terrain

(number of vertices –1 in each dimension), and then dividing this figure by how many quads will make up each dimension of a single mesh. This tells us how many meshes we will need to subdivide the terrain into along each dimension. In our case:

BlocksWide = (256)/16 = 16
BlocksHigh = (256)/16 = 16

Our terrain will thus be a grid of 16x16 meshes, where each mesh is a grid of 17x17 vertices forming 16x16 quads per mesh.

In this next line of code we set up an imaginary light vector. There is no actual light in our scene, but this vector will be used in the color calculations for determining a diffuse color for each vertex. You can think of this as a standard unit length vector just like a face normal. Instead of describing which way a polygon is facing, it is describing which direction an imaginary light is shining. We will use this approach to fake some static lighting. We cover true DirectX Graphics lighting in Chapter 5.

```
    D3DXVECTOR3 VertexPos, LightDir = D3DXVECTOR3( 0.650945f, -0.390567f, 0.650945f );
```

If we imagine the positive world Z-axis as north and the positive world X-axis as east, this vector is pointing northeast and down slightly. This will simulate the direction the sun may shine on our scene just as it is about to set.

Next we need to count how many indices each terrain mesh will need. Typically the index count needed for a triangle strip is NumberOfTriangles + 2. Think of two quads for example; we would need six indices (two rows of three) and as there are two triangles to a single quad, four triangles in total. We also need to take into account that every row but the last one will need an extra index to create the three degenerate triangles discussed in the text.

```
    // Calculate IndexCount....
    //(Number required for quads) + (Extra Degenerates verts --
    //                               one per quad row except last))
    ULONG IndexCount  = ((BlockWidth * 2) * QuadsHigh) + ( QuadsHigh - 1 );
```

Now we can calculate how many primitives these indices will create. We will need to know this for our DrawIndexedPrimitive call.

```
    //Calculate Primitive Count
    //((Number of quads ) * 2) + (3 degenerate tris per quad row except last)
    m_nPrimitiveCount = ((QuadsWide * QuadsHigh) * 2) + ((QuadsHigh - 1) * 3);
```

The CTerrain::AddMesh call has already created an array of CMeshes at this point, but they are uninitialized. So we will loop through each mesh in the array and tell it how much space it will need to reserve in its temporary arrays to hold the vertex and index data we are about to add.

```
    // Loop through and generate the mesh data
    for ( z = 0; z < BlocksHigh; z++ )
    {
        for ( x = 0; x < BlocksWide; x++ )
        {
```

```
            CMesh * pMesh = m_pMesh[ x + z * BlocksWide ];

            // Allocate all the vertices & indices required for this mesh
            if ( pMesh->AddVertex( BlockWidth * BlockHeight ) < 0 ) return false;
            if ( pMesh->AddIndex( IndexCount ) < 0 ) return false;
```

We call the CMesh::AddVertex function to reserve enough space for 17x17 vertices. We also reserve the correct number of indices using the **IndexCount** value just calculated.

Our next goal is to loop through the rows and columns for the current mesh and fill in the vertex buffer. The CMesh initially stores its vertices in a temporary array. Once the vertices and the indices have been added we will call CMesh->BuildBuffers to build the vertex and index buffer from these temporary arrays. Filling in the vertex data is much easier than the index buffer. We simply create the meshes one row at a time, where each row has its vertices specified left to right.

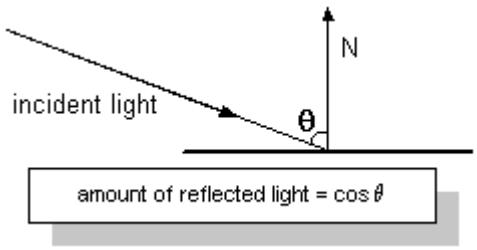
```
            // Calculate Vertex Positions
            Counter = 0;
            StartX  = x * (BlockWidth  - 1);
            StartZ  = z * (BlockHeight - 1);
            for ( vz = StartZ; vz < StartZ + BlockHeight; vz++ )
            {
                for ( vx = StartX; vx < StartX + BlockWidth; vx++ )
                {
                    // Calculate and Set The vertex data.
                    pMesh->m_pVertex[ Counter ].x = (float)vx * m_vecScale.x;
                    pMesh->m_pVertex[ Counter ].z = (float)vz * m_vecScale.z;

                    float t = (float)m_pHeightMap[ vx + vz * m_nHeightMapWidth ]
                    pMesh->m_pVertex[ Counter ].y = t * m_vecScale.y;

                    Counter++;
                } // Next Vertex Column
            } // Next Vertex Row
```

We loop through the 17 rows and 17 columns of the current mesh and use the StartX and StartY variables to create the vertex values relative to the mesh to which they belong. For example, if we are generating the mesh in the third column of the first row, the X component of the first vertex in that mesh will be X (which equals 2 because this is the third block) * block width - 1 (which is 16). So the first vertex in this mesh will have an X component of 32 (the start of the third block). The same calculation is done for the Z component using the current row of the block we are calculating. Once we have the X and Z values offset properly into the current column and row for the mesh, we multiply them by their respective components in the scale vector. The Y component of each vertex is pulled from the height map. This value is scaled by its respective component in the scale vector.

Simple Vertex Lighting

To understand how to calculate the diffuse color at each terrain vertex, we will take a brief detour to discuss some basic lighting concepts. These concepts will be revisited again in much more detail in Chapter 5.

Lambert's Law states that for an ideal diffuse surface, the intensity of the reflected light is proportional only to the cosine of the angle between the surface normal and normalized vector from the point to the light source.



If we assume that N describes a surface normal vector and L describes a normalized vector from the surface point P to the light source S:

$$L = | S - P |$$

As we discovered in Chapter 1, we can calculate the cosine of the angle between two unit length vectors using the dot product:

$$cos\theta = N \cdot L$$

To determine the final intensity at a point, we scale the light intensity by this value:

$$I_{point} = I_{light} \ (N \cdot L)$$

We can use values in the range of [0.0, 1.0] for each component of an RGB color to define a set of diffuse reflectance coefficients M_{diff_refl} for use in determining a final color for the surface point P_{color}. These coefficients are based on the properties of the surface material.

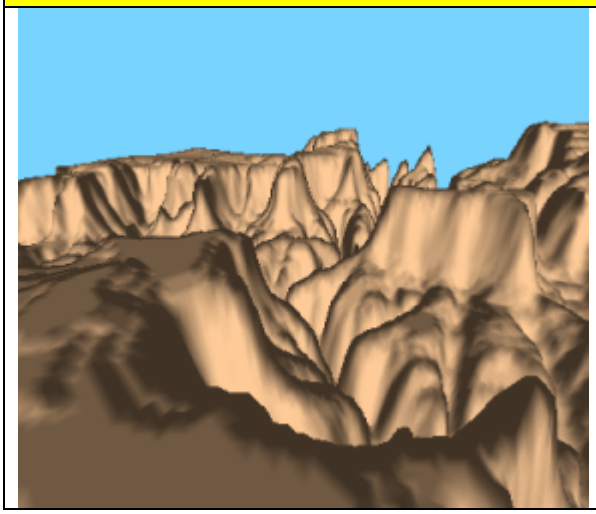
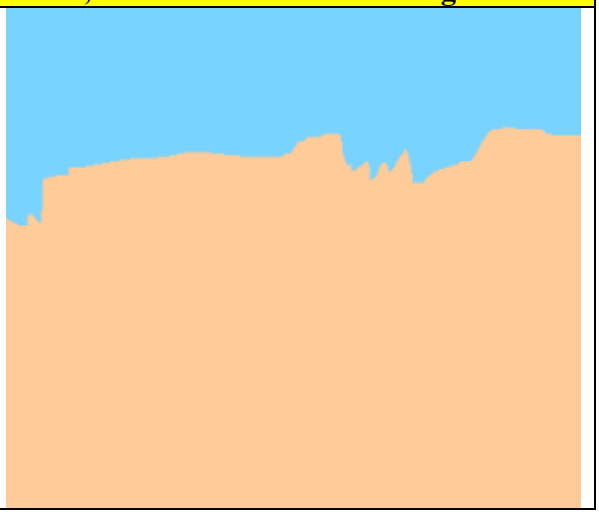
$$P_{color} = M_{diff\_refl} \ I_{point}$$

Combining the equations we get:

$$P_{color} = M_{diff\_refl} \ I_{light} \ (N \cdot L)$$

Like its parent surface, a vertex can also store a normalized 3D vector describing its own orientation. For a single triangle, each vertex normal would be the same as the face normal since the points are all co-planar. If a vertex is shared by two or more triangles, the normals of those surfaces can be averaged to produce a final value for the vertex. We will cover vertex normals in more detail in Chapter 5.

So, assuming that N in the above equation now represents a vertex orientation vector, we can simply assign our terrain its diffuse reflectance properties (in our case we will choose a brown color for a default) and determine a lighting effect for each vertex.

Different shades of the color brown at each vertex.	**Constant color: All vertices are the same color, therefore so are the triangles.**
	

The following code uses a function called GetHeightMapNormal to generate a vertex normal for a given pixel in the height map. We will actually calculate the normal for all four vertices in the current quad and average them so that we get a smoother color distribution across the vertices. Since the vertex normal will be used to help generate the diffuse color of the vertex, vertices that are next to each other but have very different normals would create abrupt changes in color from vertex to vertex. By averaging the four normals in the neighborhood of each vertex we can smooth out the color transitions.

The next section of the function loops through each vertex, and for each vertex calculates the dot product between the unit length light direction vector, and the normals of the neighboring vertices in the height map that are calculated in the GetHeightMapNormal function. Once we have the cosine of the angle between these vertex normals and the light source, we average the angle so that the angle roughly describes the cosine between the current quad direction vector (much like a face normal) and the light vector.

Note: The code that follows will make more complete sense when we cover the helper function called CTerrain::GetHeightMapNormal in a later section.

```
            // Calculate vertex color
            Counter = 0;
            for ( vz = StartZ; vz < StartZ + BlockHeight; vz++ )
            {
                for ( vx = StartX; vx < StartX + BlockWidth; vx++ )
                {
                    // Retrieve vertex position
                    VertexPos = (D3DXVECTOR3&)pMesh->m_pVertex[ Counter ];

                    // Calculate vertex colour scale
                    float fRed = 1.0f, fGreen = 0.8f, fBlue = 0.6f, fScale = 0.25f;
```

We define three floats (fRed, fGreen, and fBlue) to store our diffuse reflectance coefficients. We specify a base reflectance for the terrain that has 100% red intensity, 80% green intensity and 25% blue intensity. The result (given our white light source) is the light brown color shown in the image above.

```
                    // Generate average scale (for diffuse lighting calc)
                    fScale  = D3DXVec3Dot( &GetHeightMapNormal( vx, vz ), &LightDir);
                    fScale += D3DXVec3Dot( &GetHeightMapNormal( vx + 1, vz ), &LightDir);
                    fScale += D3DXVec3Dot( &GetHeightMapNormal( vx + 1, vz + 1 ),
                                                                        &LightDir);
                    fScale += D3DXVec3Dot( &GetHeightMapNormal( vx, vz + 1 ), &LightDir);
                    fScale /= 4.0f;
```

Four vertex normals are created (one for each vertex in the quad region) and the cosine of the angle between these vectors and the light direction vector is calculated and accumulated into fScale and then averaged.

Next, we adjust the scale value to ensure that every vertex has at least some lighting (even if they are facing completely away from the light source) by adding 0.05 to the scale value and then clamp the result to min and max values:

```
                    // Increase Saturation
                    fScale += 0.05f;

                    // Clamp colour saturation
                    if ( fScale > 1.0f ) fScale = 1.0f;
                    if ( fScale < 0.25f ) fScale = 0.25f;
```

We now have a scale value between 0.25 and 1.0, which describes how much to scale the base color components based on the orientation of the vertex and the light source. Next we use the D3DCOLOR_COLORVALUE macro, which accepts four floats that describe an RGBA color with components between 0.0 and 1.0 and returns a DWORD where each component is mapped to the [0, 255] range. We store this DWORD in our vertex structure as the diffuse color.

Finally we scale the base color by the averaged cosine of the angle between the current quad and the light direction vector. This will scale the color of the vertex based on the quad's orientation with respect to the light vector. This is simple but effective diffuse lighting formula used by many 3D rendering engines.

```
         // Store Colour Value
         pMesh->m_pVertex[ Counter ].Diffuse = D3DCOLOR_COLORVALUE(
                    fRed  * fScale,
                    fGreen * fScale,
                    fBlue  * fScale,
                    1.0f );

         Counter++;

        } // Next Vertex Column

   } // Next Vertex Row
```

Note that a light source color was never specified in the above code because our demo assumes a white light source (1.0 for all components). We also set the alpha value of the color to 1.0 in the call to D3DCOLOR_COLORVALUE. We will use alpha values later in Chapter 7, but until then we will continue to set them to 1.0 (completely opaque).

We now have our current mesh with its vertex array complete. Each vertex has a color that is some shade of the base vertex color. Now it is time to add the indices. Remember that we will insert a duplicate index to the first vertex at the start of each row (except the first row) to create our degenerate triangles.

```
Counter = 0;

// Calculate the indices for the terrain block tri-strip
for ( vz = 0; vz < BlockHeight - 1; vz++ )
{
   // Is this an odd or even row ?
   if ( (vz % 2) == 0 )
   {
     for ( vx = 0; vx < BlockWidth; vx++ )
     {
          // Force insert winding order switch degenerate ?
          if ( vx == 0 && vz > 0 )
             pMesh->m_pIndex[ Counter++ ] = (USHORT)(vx + vz * BlockWidth);

          // Insert next two indices
          pMesh->m_pIndex[ Counter++ ] = (USHORT)(vx + vz * BlockWidth);
          pMesh->m_pIndex[ Counter++ ] = (USHORT)((vx + vz * BlockWidth) + BlockWidth);

        } // Next Index Column

     } // End if even row
   else
   {
     for ( vx = BlockWidth - 1; vx >= 0; vx--)
     {
        // Force insert winding order switch degenerate ?
        if ( vx == (BlockWidth - 1) )
           pMesh->m_pIndex[ Counter++ ] = (USHORT)(vx + vz * BlockWidth);

        // Insert next two indices
        pMesh->m_pIndex[ Counter++ ] = (USHORT)(vx + vz * BlockWidth);
```

```
            pMesh->m_pIndex[ Counter++ ] = (USHORT)((vx + vz * BlockWidth) + BlockWidth);
        } // Next Index Column

    } // End if odd row

} // Next Index Row
```

We start off by doing the first row (row[0]) of indices. This is an even row. We move along the width of the row adding indices for the current vertex and the vertex above it (below it in image space) just as we saw in the text. When we get to the end of the row, the **vz** loop increments and we enter the odd row **vx** loop in **vz's** next iteration (the else statement). This starts adding pairs of vertices in reverse order from right to left. Notice that the first thing it does is insert the duplicate index into the first vertex of that row. We then add the row vertices as usual. The duplicate index creates the three degenerates described in the lesson. Once we reach the end of that row (remember that we are adding odd rows from right to left and even rows from to left to right) the **vz** loop increments again to take us to the third row (row[2]). The order switches again and we start adding pairs of vertices from the start of this row, working left to right as we did in the first row. Because this is not the first row, the duplicate index is added to the first vertex in this even row causing the three degenerate triangles again on the left side. We repeat this procedure until all rows are complete. When the loops exit, the mesh has completely filled its vertex and index arrays.

The final step is construction of the vertex and index buffers using the CMesh::BuildBuffer function discussed previously.

```
            // Instruct mesh to build buffers
            if ( FAILED(pMesh->BuildBuffers( m_pD3DDevice, m_bHardwareTnL )) )
                 return false;
```

We repeat this process for every mesh in the terrain (16x16 meshes in our example).

```
        } // Next Block Column
    } // Next Block Row

    // Success!
    return true;
}
```

Below we see the complete CTerrain::BuildMeshes function without any interruptions.

```
bool CTerrain::BuildMeshes( )
{
    long x, z, vx, vz, Counter, StartX, StartZ;
    long BlocksWide = (m_nHeightMapWidth  - 1) / QuadsWide;
    long BlocksHigh = (m_nHeightMapHeight - 1) / QuadsHigh;

    D3DXVECTOR3 VertexPos, LightDir = D3DXVECTOR3( 0.650945f, -0.390567f, 0.650945f );

    ULONG IndexCount  = ((BlockWidth * 2) * QuadsHigh) + ( QuadsHigh - 1 );
    m_nPrimitiveCount = ((QuadsWide * QuadsHigh) * 2) + ((QuadsHigh - 1) * 3);
```

```
    // Loop through and generate the mesh data
    for ( z = 0; z < BlocksHigh; z++ )
    {
        for ( x = 0; x < BlocksWide; x++ )
        {
            CMesh * pMesh = m_pMesh[ x + z * BlocksWide ];

            // Allocate all the vertices & indices required for this mesh
            if ( pMesh->AddVertex( BlockWidth * BlockHeight ) < 0 ) return false;
            if ( pMesh->AddIndex( IndexCount ) < 0 ) return false;

            // Calculate Vertex Positions
            Counter = 0;
            StartX  = x * (BlockWidth  - 1);
            StartZ  = z * (BlockHeight - 1);
            for ( vz = StartZ; vz < StartZ + BlockHeight; vz++ )
            {
                for ( vx = StartX; vx < StartX + BlockWidth; vx++ )
                {
                    // Calculate and Set The vertex data.
                    pMesh->m_pVertex[ Counter ].x = (float)vx * m_vecScale.x;
                    pMesh->m_pVertex[ Counter ].y = \
                                (float)m_pHeightMap[vx+vz*m_nHeightMapWidth]*m_vecScale.y;
                    pMesh->m_pVertex[ Counter ].z = (float)vz * m_vecScale.z;
                    Counter++;

                }
            }

            // Calculate vertex lighting
            Counter = 0;
            for ( vz = StartZ; vz < StartZ + BlockHeight; vz++ )
            {
                for ( vx = StartX; vx < StartX + BlockWidth; vx++ )
                {
                    // Retrieve vertex position
                    VertexPos = (D3DXVECTOR3&)pMesh->m_pVertex[ Counter ];

                       // Calculate vertex colour scale
                       float fRed = 1.0f, fGreen = 0.8f, fBlue = 0.6f, fScale = 0.25f;

                       // Generate average scale (for diffuse lighting calc)
                        fScale  =  D3DXVec3Dot( &GetHeightMapNormal( vx, vz ), &LightDir);
                        fScale += D3DXVec3Dot( &GetHeightMapNormal( vx + 1, vz ),
                                               &LightDir);
                        fScale += D3DXVec3Dot( &GetHeightMapNormal( vx + 1, vz + 1 ),
                                               &LightDir);
                        fScale += D3DXVec3Dot( &GetHeightMapNormal( vx, vz + 1 ),
                                               &LightDir);
                        fScale /= 4.0f;

                        // Increase Saturation
                        fScale += 0.05f;

                        // Clamp colour saturation
                        if ( fScale > 1.0f ) fScale = 1.0f;
                        if ( fScale < 0.25f ) fScale = 0.25f;

                        // Store Colour Value
```

```
                    pMesh->m_pVertex[ Counter ].Diffuse = D3DCOLOR_COLORVALUE(fRed*fScale,
                                                                             fGreen*fScale,
                                                                             fBlue*fScale,
                                                                             1.0f );

                    Counter++;

                } // Next Vertex Column

            } // Next Vertex Row

            Counter = 0;

            // Calculate the indices for the terrain block tri-strip
            for ( vz = 0; vz < BlockHeight - 1; vz++ )
            {
                // Is this an odd or even row ?
                if ( (vz % 2) == 0 )
                {
                    for ( vx = 0; vx < BlockWidth; vx++ )
                    {
                        // Force insert winding order switch degenerate ?
                        if ( vx == 0 && vz > 0 )
                          pMesh->m_pIndex[ Counter++ ] = (USHORT)(vx + vz * BlockWidth);

                        // Insert next two indices
                        pMesh->m_pIndex[ Counter++ ] = (USHORT)(vx + vz * BlockWidth);
                        pMesh->m_pIndex[ Counter++ ] = (USHORT)((vx + vz * BlockWidth) +
                                                               BlockWidth);
                    } // Next Index Column
                } // End if even row
                else
                {
                    for ( vx = BlockWidth - 1; vx >= 0; vx--)
                    {
                        // Force insert winding order switch degenerate ?
                        if(vx == (BlockWidth - 1))
                           pMesh->m_pIndex[Counter++] = (USHORT)(vx + vz * BlockWidth);

                        // Insert next two indices
                        pMesh->m_pIndex[ Counter++ ] = (USHORT)(vx + vz * BlockWidth);
                        pMesh->m_pIndex[ Counter++ ] = (USHORT)((vx + vz * BlockWidth) +
                                                               BlockWidth);

                    } // Next Index Column
                } // End if odd row
            } // Next Index Row

            // Instruct mesh to build buffers
            if ( FAILED(pMesh->BuildBuffers( m_pD3DDevice, m_bHardwareTnL )) )
                        return false;

        } // Next Block Column

    } // Next Block Row

    // Success!
    return true;
}
```

The hard part is now over. Please make sure that you take the time to understand how the whole process works. Learning how to represent a grid of quads as a triangle strip will prove to be very useful to you in your programming future.

CTerrain::GetHeightmapNormal

The GetHeightMapNormal function takes as input the location of a pixel in the height map (which is also a vertex in the terrain) and returns a unit length normal vector describing the direction in which the pixel (vertex) is facing in world space. Notice that the parameters are labeled x and z instead of x and y. In reality we are passing in the **x:y** coordinates of the pixel in the height map to generate the normal for a vertex stored at **x:z** in world space. Just remember that the image space Y-axis is the world space Z-axis.

```
D3DXVECTOR3 CTerrain::GetHeightMapNormal( ULONG x, ULONG z )
{
     D3DXVECTOR3 Normal, Edge1, Edge2;
     ULONG       HMIndex, HMAddX, HMAddZ;
     float       y1, y2, y3;

     // Make sure we are not out of bounds
     if ( x < 0.0f || z < 0.0f || x >= m_nHeightMapWidth || z >= m_nHeightMapHeight )
          return D3DXVECTOR3(0.0f, 1.0f, 0.0f);
```

The first thing we do is make sure that the image coordinates passed are not outside the bounds of the image. If the point is out of bounds, we simply return a vector aligned with the world space Y-axis. This vector is a good generic normal for a terrain if the worst comes to the worst. Provided that valid coordinates have been specified, we now need to know what the offset of that pixel is in our height map array. Remember that the height map is a one-dimensional linear array. All of the rows are arranged in memory one after another. To calculate the index of the desired pixel in the array, we multiply the row number by the number of pixels that are in a row and then add the column number.

```
    // Calculate the index in the heightmap array
    HMIndex = x + z * m_nHeightMapWidth;
```

If the image map was 10x10 pixels in size and we specified a coordinate of (3, 7) this would be calculated as:

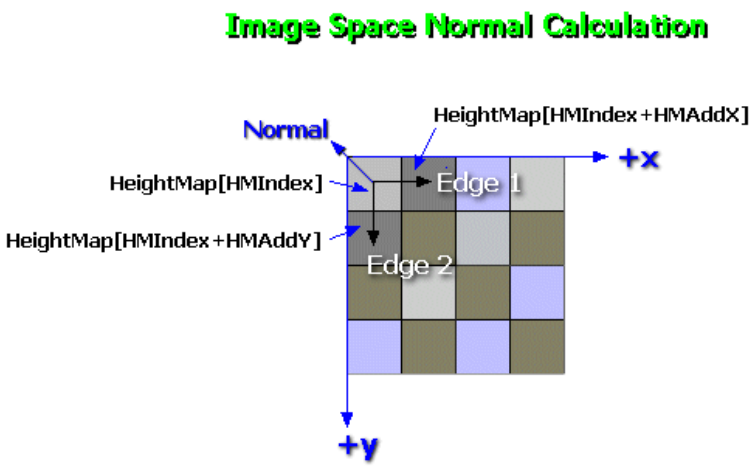
3 + (7*10) = 74

So array element 73 in the height map array would be the height for pixel (3, 10) in the image.

When we discussed face normals in Chapter 1, we realized that when we have three vertices that define a triangle, we can calculate the direction that triangle is facing by creating two vectors using two of its edges and then performing a cross product operation on them. The result is a vector that is perpendicular to the other two. This is the face normal.

Although we do not actually have a triangle as such, we do have the height map. Every pixel in the height map is essentially a virtual vertex. Therefore, we can create two edge vectors to form a virtual triangle and perform the cross product on these two vectors to get the pixel normal.

In order to create the two vectors, we will need three vertices. We already have the index of the first vertex in the height map (HMIndex) based on the pixel coordinate passed in. We can use the pixel immediately to the right of it and the pixel immediately below it as the second and third vertices. We now have three vertices with which to create two edge vectors. The following image shows a magnified view of the pixels at the top left corner of the height map. If the coordinates passed in were (0, 0) then we wish to calculate the normal for the first pixel at the top left corner (HMIndex). We also use the pixel immediately to the right (HMIndex+HMAddX) and the pixel below (HMIndex+HMAddy).



If the coordinate passed in is for a pixel at either the far right edge of the image or at the bottom row of the image, we reverse direction and use the pixel to the right or above it respectively. We control this with the HMAddX and HMAddY variables as shown below.

```
    // Calculate the number of pixels to add in either direction to
    // obtain the best neighboring heightmap pixel.
    if ( x < (m_nHeightMapWidth - 1))  HMAddX = 1;
    else                               HMAddX = -1;

    if ( z < (m_nHeightMapHeight - 1)) HMAddZ = m_nHeightMapWidth;
    else                               HMAddZ = -(signed)m_nHeightMapWidth;
```

We add these offsets to HMindex and retrieve the neighbouring pixels:

```
    // Get the three height values
    y1 = (float)m_pHeightMap[HMIndex] * m_vecScale.y;
    y2 = (float)m_pHeightMap[HMIndex + HMAddX] * m_vecScale.y;
    y3 = (float)m_pHeightMap[HMIndex + HMAddZ] * m_vecScale.y;
```

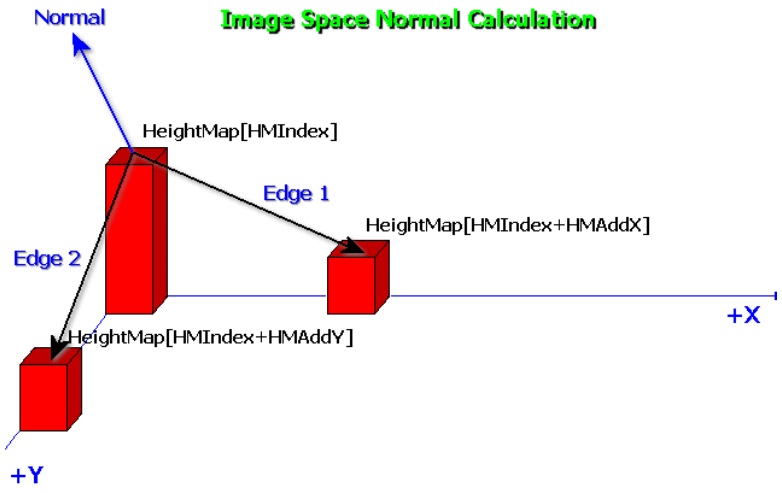
The prior code samples the three L-Shaped height values from the height map and scales them by the Y component of our scale vector so that we can create two edge vectors with the correct scale of our terrain.

```
    // Calculate Edges
    Edge1 = D3DXVECTOR3( m_vecScale.x, y2 - y1, 0.0f );
    Edge2 = D3DXVECTOR3( 0.0f, y3 - y1, m_vecScale.z );
```

The scale vector we used in our demo is (8, 2, 8). If we passed in image coordinate (0, 0) and pixel (0, 0) had a value of 20 in the height map and pixel (0, 1) has a height of 40, we would create the first edge vector:

(m_vecScale.x , 40-20 * m_vecScale.y, m_vecScale.z) = Edge 1 = (8 , 40 , 8)

This vector accurately describes the slope from pixel/vertex (0, 0) to pixel/vertex (0, 1) as shown below. The following image is a 3D representation of our height map. It is as if we lowered ourselves into the image map and were looking at the top left corner of the image.



Once we have the two edge vectors we can perform a cross product to generate a vector that is perpendicular to the two edge vectors and then normalize it.

```
    // Calculate Resulting Normal
    D3DXVec3Cross( &Normal, &Edge1, &Edge2);
    D3DXVec3Normalize( &Normal, &Normal );

    // Return it.
    return Normal;
}
```

Here is the complete code to the function without interruption:

```
D3DXVECTOR3 CTerrain::GetHeightMapNormal( ULONG x, ULONG z )
{
    D3DXVECTOR3 Normal, Edge1, Edge2;
    ULONG       HMIndex, HMAddX, HMAddZ;
    float       y1, y2, y3;

    // Make sure we are not out of bounds
    if ( x < 0.0f || z < 0.0f || x >= m_nHeightMapWidth || z >= m_nHeightMapHeight )
         return D3DXVECTOR3(0.0f, 1.0f, 0.0f);

    // Calculate the index in the heightmap array
    HMIndex = x + z * m_nHeightMapWidth;

    // Calculate the number of pixels to add in either direction to
    // obtain the best neighbouring heightmap pixel.
    if ( x < (m_nHeightMapWidth - 1))  HMAddX = 1;
    else HMAddX = -1;

    if ( z < (m_nHeightMapHeight - 1)) HMAddZ = m_nHeightMapWidth;
    else HMAddZ = -(signed)m_nHeightMapWidth;

    // Get the three height values
    y1 = (float)m_pHeightMap[HMIndex] * m_vecScale.y;
    y2 = (float)m_pHeightMap[HMIndex + HMAddX] * m_vecScale.y;
    y3 = (float)m_pHeightMap[HMIndex + HMAddZ] * m_vecScale.y;

    // Calculate Edges
    Edge1 = D3DXVECTOR3( m_vecScale.x, y2 - y1, 0.0f );
    Edge2 = D3DXVECTOR3( 0.0f, y3 - y1, m_vecScale.z );

    // Calculate Resulting Normal
    D3DXVec3Cross( &Normal, &Edge1, &Edge2);
    D3DXVec3Normalize( &Normal, &Normal );

    // Return it.
    return Normal;
}
```

CTerrain::Render

Recall that our code framework repeatedly calls the FrameAdvance function to render the frame. This function will in turn call CTerrain::Render. Our task here is actually quite straightforward. The function will simply loop through each mesh, sets its vertex and index buffer and call DrawIndexedPrimitive to render the entire mesh as a single indexed triangle strip.

```
void CTerrain::Render( )
{
    // Validate parameters
    if( !m_pD3DDevice ) return;

    // Render Each Mesh
    for ( ULONG i = 0; i < m_nMeshCount; i++ )
    {
        // Set the stream sources
```

```
        m_pD3DDevice->SetStreamSource(0, m_pMesh[i]->m_pVertexBuffer, 0, sizeof(CVertex));
        m_pD3DDevice->SetIndices( m_pMesh[i]->m_pIndexBuffer );

        // Render the vertex buffer
        m_pD3DDevice->DrawIndexedPrimitive( D3DPT_TRIANGLESTRIP, 0, 0,
                                            BlockWidth * BlockHeight,
                                            0, m_nPrimitiveCount );

    } // Next Mesh
}
```

Lab Project 3.3: Dynamic Vertex Buffers

Our previous demonstrations have used static vertex buffers because our data never needed to be modified after it was created. There are circumstances however when we will want to manipulate the vertices in a buffer relatively frequently (like once per frame for example). In these cases, we will need to utilize dynamic vertex buffers.

Imagine for example if we had a function that returns only the visible polygons from the current position of the camera. The call might look something like this:

```
VisibilitySystem->GetVisibleTriangles(&mtxView,  pVertexBuffer);
```

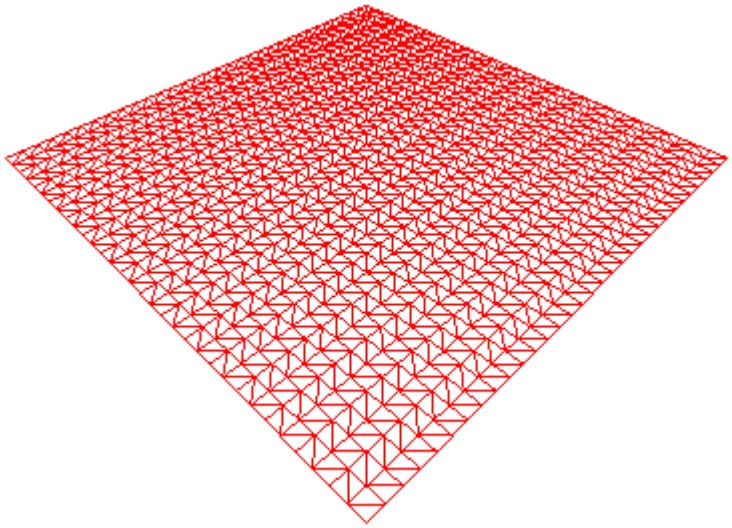
If the game world was made up of hundreds of thousands of polygons, we might not want to store them all in a video memory vertex buffer. This would take up a good deal of space that may be better reserved for textures and other important resources. We could store the level in a system memory vertex buffer but rendering from system memory vertex buffers is a slow process.

The best bet may be to store the level in a standard application memory array that we can access quickly without the overhead of locking buffers. In that case, the visibility system could collect only the visible polygons and throw them into a dynamic vertex buffer for rendering. Writing to a dynamic vertex buffer is typically fast. It can be stored in video memory without taking up too much room since it will only hold a relatively low number of triangle vertices each frame. We would repeat the process every frame. The previous frame's vertex buffer would be flushed and the visibility system would fill the dynamic buffer for the next frame (which might contain a different subset of polygons if the camera is in a different position). The dynamic vertex buffer in this case essentially provides a vertex caching system where the application can add the polygons needed for the frame, render, flush and repeat.

Animation presents another common use for a dynamic vertex buffer and this will be the subject of our final demonstration for this lesson. Our project will look at a technique that can be used to create a simple wave effect. The mesh we use will be a flat surface made up of rows of quads much like our terrain arrangement. We can alter the positions of the vertices every frame to create the effect of ripples or waves in the mesh. We can lock the dynamic buffer using the D3DLOCK_DISCARD and D3DLOCK_NOOVERWRITE flags to inform the driver that it can either issue us a new buffer to write our vertices or that it can go on rendering from the buffer because we are not going to overwrite

any data. This same technique can be applied to other circumstances that involve such motion (wind blown flags for example).

The image above is taken from the final demonstration application that we will create in this chapter. The application begins with a flat sheet (patch) of quads. This sheet will be stored in a single dynamic vertex buffer. There will be 33 x 33 vertices so the triangle strip will consist of 32 rows of 32 quads.



We already know how to create one of these vertex sheets because we did it in our last project. This time however we will not generate the vertex buffer at application startup. Instead, we will build the vertex buffer on the fly each frame in the AnimateObjects function. This function will fill the vertex buffer with 33x33 vertices using a sine wave to adjust the height of each vertex. This will propagate the change over distance and time.

Note that since we know in advance how the vertices will be arranged in the vertex buffer, we can build the index buffer at application startup, even though the vertex buffer is not built until later.

Changing the values or rewriting the height value of a vertex does not change which triangles it belongs to.

The CGameApp Class

The entire patch will be stored using one vertex buffer and one index buffer so we added only a few new members to the CGameApp class. No extra classes are used in this demo (like CTerrain for example) since it is very simple. The code is almost all contained in the CGameApp.cpp file.

Changes to CGameApp class member variables

```
D3DXMATRIX               m_mtxView;          // View Matrix
D3DXMATRIX               m_mtxProjection;    // Projection matrix
LPDIRECT3DVERTEXBUFFER9  m_pVertexBuffer;    // Vertex Buffer to be Rendered
LPDIRECT3DINDEXBUFFER9   m_pIndexBuffer;     // Index Buffer to be Rendered
ULONG                    m_nIndexCount;      // Number of indices stored.
bool                     m_bAnimation;       // Mesh Animation enabled / disabled
```

This demo has no camera movement, so there is no need to include our CCamera class. The view and projection matrices will be created at application startup and never touched again. The index buffer will be filled once at application startup and never touched again, so we will want to create a static index buffer. The vertex buffer will be created at application startup also but it will not be filled until just before rendering each frame. Finally, we also have a boolean variable which allows the user to stop/start the animation of the vertices.

The first framework function that has some changes that require explanation is BuildObjects. Since this is the function that assembles objects and meshes for all of our demos, it will almost always be completely different for each demo we write.

CGameApp::BuildObjects

The code for building the index buffer is exactly the same as the last demo. The vertex buffer creation code now creates a dynamic vertex buffer instead of a static one, but we will not fill it here.

First we determine whether we are using a hardware or software vertex-processing device. This allows us to set appropriate D3DUSAGE flags when we create the vertex buffer. We then call the IDirect3DDevice9::CreateVertexBuffer function as shown below.

```
bool CGameApp::BuildObjects()
{
    HRESULT  hRet;
    CVertex *pVertex = NULL;
    USHORT  *pIndex  = NULL;
    ULONG    ulUsage = D3DUSAGE_WRITEONLY;
    long     vx, vz;
```

```
    // Seed the random number generator
    srand( timeGetTime() );

    // Release previously built objects
    ReleaseObjects();

    // Build our buffers usage flags (i.e. Software T&L etc)
    VERTEXPROCESSING_TYPE vp = m_D3DSettings.GetSettings()->VertexProcessingType;
    if ( vp != HARDWARE_VP && vp != PURE_HARDWARE_VP )
        ulUsage |= D3DUSAGE_SOFTWAREPROCESSING;

    // Create our vertex buffer
    m_pD3DDevice->CreateVertexBuffer(sizeof(CVertex) * (BlockWidth * BlockHeight),
                                                          D3DUSAGE_DYNAMIC | ulUsage,
                                                          D3DFVF_XYZ | D3DFVF_DIFFUSE,
                                                          D3DPOOL_DEFAULT, &m_pVertexBuffer,
                                                          NULL );
```

BlockWidth and BlockHeight are constants with values of 33. This means that we are creating a vertex buffer with enough room for 33x33 vertices. Note that we use the D3DUSAGE_DYNAMIC flag when creating the vertex buffer and that we use the D3DPOOL_DEFAULT pool instead of the D3DPOOL_MANAGED pool used in previous demos. We must make sure that we remember to rebuild the vertex buffer in response to the device becoming lost because D3DPOOL_DEFAULT resources are not automatically rebuilt when the device is reset. We can do this simply by calling the BuildObjects function again. This approach works because the function always calls the ReleaseObjects function prior to building its objects to clean up any outstanding resources.

The next section of code builds the index data. First we calculate the index count. QuadsHigh is set to 32 because 33x33 vertices create 32x32 quads. As with the previous demo, we also add an extra index for all rows but the first to create the three degenerate triangles necessary to render the single strip.

```
    // Calculate IndexCount....
    //( Number required for quads ) + ( Extra Degenerates (one per quad row except last) )
    m_nIndexCount = ((BlockWidth * 2) * QuadsHigh) + ( QuadsHigh - 1 );

    m_pD3DDevice->CreateIndexBuffer( sizeof(USHORT) * m_nIndexCount,
                                     ulUsage, D3DFMT_INDEX16,
                                     D3DPOOL_DEFAULT, &m_pIndexBuffer, NULL );
```

The index buffer is locked and filled exactly as it was in the previous demonstration. The only difference is that we fill the buffer directly rather than using the temporary array approach seen last time.

```
    // Lock the index buffer (we only need to build this once in this example)
    m_pIndexBuffer->Lock( 0, sizeof(USHORT) * m_nIndexCount, (void**)&pIndex, 0 );

    // Calculate the indices for the patch block tri-strip
    for ( vz = 0; vz < BlockHeight - 1; vz++ )
    {
        // Is this an odd or even row ?
        if ( (vz % 2) == 0 )
        {
```

```
            for ( vx = 0; vx < BlockWidth; vx++ )
            {
                // Force insert winding order switch degenerate ?
                if ( vx == 0 && vz > 0 ) *pIndex++ = (USHORT)(vx + vz * BlockWidth);

                // Insert next two indices
                *pIndex++ = (USHORT)(vx + vz * BlockWidth);
                *pIndex++ = (USHORT)((vx + vz * BlockWidth) + BlockWidth);
            }
        }
        else
        {
            for ( vx = BlockWidth - 1; vx >= 0; vx--)
            {
                // Force insert winding order switch degenerate ?
                if ( vx == (BlockWidth - 1) ) *pIndex++ = (USHORT)(vx + vz * BlockWidth);

                // Insert next two indices
                *pIndex++ = (USHORT)(vx + vz * BlockWidth);
                *pIndex++ = (USHORT)((vx + vz * BlockWidth) + BlockWidth);
            }
        }
    } // Next Index Row

    // Unlock the index buffer
    if ( FAILED(m_pIndexBuffer->Unlock()) ) return false;

    // Force a rebuild of the vertex data
    AnimateObjects();

    // Success!
    return true;
}
```

Notice that we make a call to CGameApp::AnimateObjects before exiting. This function is usually called from CGameApp:FrameAdvance in our main render loop to build the world matrices for our objects. We only have one object in this demo and the vertices are in world space, so no world matrix is required for the patch. Instead, AnimateObjects locks the dynamic the vertex buffer (discarding any previous contents), and refills the vertex data each frame. When the vertices are added, the Y component will be adjusted each time. This causes the vertices to move in a ripple like pattern. The reason we call the function here in BuildObjects is to force an initial build of the vertex buffer before we start the main rendering loop.

The SetupGameState function is very simple in this project. It uses the D3DXMatrixLookAtLH function to build a view matrix and makes sure that m_nAnimation is initially set to true so that our vertices are animated each frame.

```
void CGameApp::SetupGameState()
{
    // Setup Default Matrix Values
    D3DXMatrixIdentity( &m_mtxView );
```

```
    // Set our cameras look at
    D3DXMatrixLookAtLH( &m_mtxView,
                        &D3DXVECTOR3(-24.0f, 90.0f, -24.0f ),
                        &D3DXVECTOR3(-23.0f, 88.5f, -23.0f ),
                        &D3DXVECTOR3(  0.0f,  1.0f,   0.0f ) );

    // Enable animation
    m_bAnimation = true;

    // App is active
    m_bActive    = true;
}
```

The SetupRenderState function initializes the projection matrix, device render states, and flexible vertex format. It attaches the vertex buffer to stream zero and binds the index buffer using SetIndices. Finally, it sends the view and projection matrices to the device and the transformation pipeline is ready to be used.

```
void CGameApp::SetupRenderStates()
{
    // Set up new perspective projection matrix
    float fAspect = (float)m_nViewWidth / (float)m_nViewHeight;
    D3DXMatrixPerspectiveFovLH( &m_mtxProjection, D3DXToRadian( 60.0f ),
                                fAspect, 1.01f, 1000.0f );

    // Setup our D3D Device initial states
    m_pD3DDevice->SetRenderState( D3DRS_ZENABLE, D3DZB_TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_DITHERENABLE,  TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_SHADEMODE, D3DSHADE_GOURAUD );
    m_pD3DDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CCW );
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, FALSE );

    // Setup our vertex FVF code
    m_pD3DDevice->SetFVF( D3DFVF_XYZ | D3DFVF_DIFFUSE );

    // Set the stream sources
    m_pD3DDevice->SetStreamSource( 0, m_pVertexBuffer, 0, sizeof(CVertex) );
    m_pD3DDevice->SetIndices( m_pIndexBuffer );

    // Setup our matrices
    m_pD3DDevice->SetTransform( D3DTS_VIEW, &m_mtxView );
    m_pD3DDevice->SetTransform( D3DTS_PROJECTION, &m_mtxProjection );
}
```

Because the patch is rendered as a single indexed triangle strip, you can probably anticipate how simple the FrameAdvance function will be in this demo. The following code shows the relevant section of the FrameAdvance function. It consists of a single call to DrawIndexedPrimitive to render the entire patch.

```
    // Animate the meshes
    if ( m_bAnimation ) AnimateObjects();

    // Clear the frame & depth buffer ready for drawing
    m_pD3DDevice->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, 0xFFFFFFFF, 1.0f, 0);
```

```
    // Begin Scene Rendering
    m_pD3DDevice->BeginScene();

    // Render the primitive
    m_pD3DDevice->DrawIndexedPrimitive( D3DPT_TRIANGLESTRIP, 0, 0,
                                        (BlockWidth * BlockHeight), 0, m_nIndexCount - 2 );

    // End Scene Rendering
    m_pD3DDevice->EndScene();

    // Present the buffer
    if ( FAILED(m_pD3DDevice->Present( NULL, NULL, NULL, NULL )) ) m_bLostDevice = true;
```

CGameApp::AnimateObjects

This is the animation routine for our ripple effect. To begin, we create a static float that is initialised to 360 the first time the function is called. This value will be decremented each time the function is called so that it runs down from 360 to 0 to be reset to 360 again at that point. This will be our means of animation. Larger decrements will result in faster waves.

```
void CGameApp::AnimateObjects()
{
    static float Distance = 360.0f;

    ULONG       x, z;
    HRESULT     hRet;
    float       fHeight;
    CVertex    *pVertex  = NULL;
    D3DXVECTOR3 vecScale = D3DXVECTOR3( 4.0f, 6.0f, 4.0f );
```

We create a scale vector like we did in the last demo because we will want to scale the vertex positions as we add them to the vertex buffer. If we did not do this, our 33x33 vertex terrain patch would be limited to a size of 33x33 units in world space. We will scale the vertex positions by 4 in the X and Z dimensions and 6 in the Y dimension (effectively making our 33x33 vertex patch 132x132 world space units).

The next piece of code is responsible for the vertex animation. All it does is subtract a scaled elapsed time value from the static Distance variable and loops back around to 360 when the value becomes less than zero.

```
    // Work out time shift
    Distance -= 5.0f * m_Timer.GetTimeElapsed();
    if (Distance < 0.0f) Distance += 360.0f;
```

Our next task is to lock the dynamic vertex buffer. We use D3DLOCK_DISCARD to inform the driver that we will be filling it with new vertex data and do not need any data from the previous frame. This is very important because if we did not specify this flag and the driver was currently rendering from this

vertex buffer, we would have to wait until it had finished before it released its claim on the buffer. This causes a stall in the pipeline and our application will sit idle until the GPU is finished with the buffer.

With this flag set, if the GPU is currently rendering from the vertex buffer we wish to lock, we will not have to wait. A new buffer pointer will be returned and we can write our vertices to this buffer at the same time the GPU is rendering from the old one. This is called *vertex buffer renaming* and happens behind the scenes.

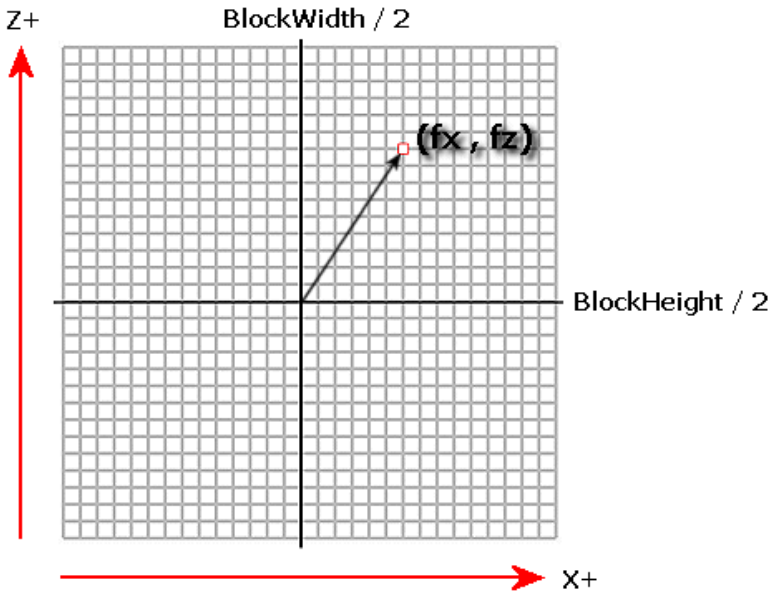
```
    // Lock the vertex buffer
    hRet = m_pVertexBuffer->Lock( 0, sizeof(CVertex) * (BlockWidth * BlockHeight),
                                  (void**)&pVertex, D3DLOCK_DISCARD );
```

Now we are going to create two nested loops so that we can loop through the rows and columns of our patch and add a vertex at each point.

```
    // Loop through each row
    for ( z = 0; z < BlockHeight; z++ )
    {
        // Loop through each column
        for ( x = 0; x < BlockWidth; x++ )
        {
            // Calculate height of the vertex
            float fx = ((BlockWidth / 2.0f) - x);
            float fz = ((BlockHeight / 2.0f) - z);
            float fDist = sqrtf(fx * fx + fz * fz) ;//+ Distance;
            fHeight = sinf(fDist) * (vecScale.y / 2.0f);
```

The above code calculates the height for vertex (X, Z) on the grid using the sine (sinf) trigonometry function. Vertex heights will be calculated using a sine wave pattern. Let us examine this a little more closely.

First we calculate fx and fz by subtracting the vertex position (X, Height , Z) from the grid center point. We now have a 2D vector relative to the center point of the patch. The black arrow in the following image is a 2D vector (fx,fz) describing the orientation and distance from the center of the terrain to grid point (X:Z).

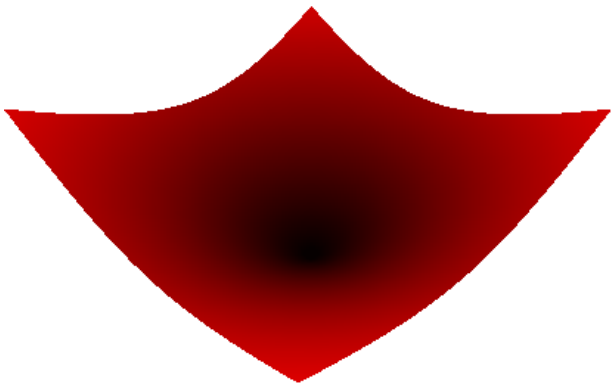


The next thing we do is calculate the length of this vector:

```
float fDist = sqrtf(fx * fx + fz * fz) + Distance;
```

Notice that we add the Distance value to the calculated vector length. The Distance variable is the animation variable that cycles over time from 360 to 0.

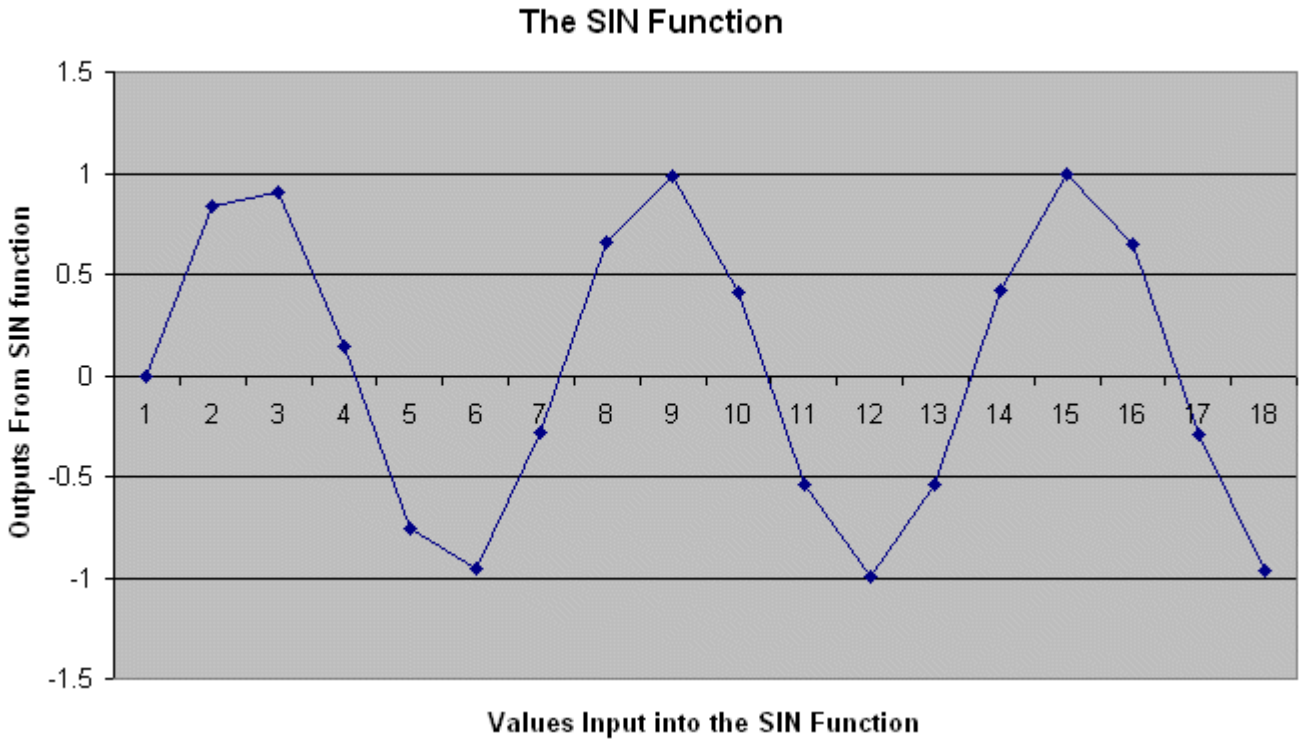
Forgetting about the Distance value for a moment, we could say that if this vector length was assigned to the vertex height (the Y component), then we can imagine that vertices further from the center have longer vectors, and are therefore higher in the 3D world. Vertices closer to the center of the grid would have lower values. If we were to render the patch in this state it would look like a dried up slice of bread: low in the middle and highest at the four corners points):



Vertex Height = Distance From Center

The sin function can now help us modify the vertex heights so that we begin to see waves.

The sin function has a range of –1 to +1. Whatever value you pass, you will always get back a float in that range. The function does not clamp the input value so that it produces an output in the [–1, 1] range. It works more like our Distance function where values that would generate an output larger than 1 for example, will be rolled over into the –1 range. The following image shows outputs from the sin function with inputs between 0 – 18.

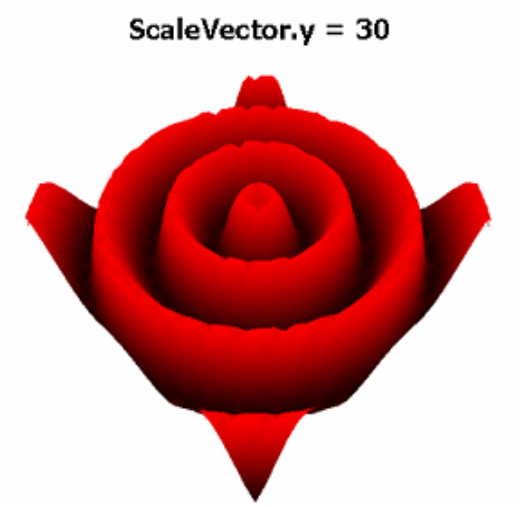


Now imagine that those input values are the distances from the center of the grid to its vertices. You can see that the output values do indeed fall into in the range of –1 to +1. These periodic values produce a wave-like result. Instead of just using the vertex vector length for its height, we can feed in the vector length to sin and get a height in the –1 to +1 range. If we want to scale the values up to be a bit to make our waves more prominent than heights of –1 to +1, we can multiply the result by the Y component of our scale vector.

```
// Calculate height of the vertex
float fx = ((BlockWidth / 2.0f) - x);
float fz = ((BlockHeight / 2.0f) - z);
float fDist = sqrtf(fx * fx + fz * fz) + Distance;
fHeight = sinf(fDist) * (vecScale.y / 2.0f );
```

We use a scale vector with a Y component of 6. This describes the length we would like a vector to be from the lowest wave position to the highest wave position. We divide the scale vector by 2 because if we did not, the result would be height values in the range of –6 to +6, which is actually a scale of 12. Since our desired scale is 6, we divide by 2 to get height values in the range of –3 to +3. By increasing the Y component of the scale vector we can make the waves much bigger.

This next image was taken with a scale vector that had a Y component of 30 (vertex heights range [-15,15]).



If we look at the figures in the sin graph, we can see that if we add a constantly decreasing or increasing variable to each value before we send it into the function, the height of each vertex would bob up and down on the sine wave. For example, imagine we have a vertex with a height of 8. On the next frame we add 1 to this amount so it becomes 9 and then in the following frame we add two and it becomes 10. Now look at the image of the sin graph again. The height of the vertex would follow the graph line from 8 to 11. If we keep incrementing the value each time, the vertex height will follow the curve of the graph. Each vertex will be at a different position along the curve at any given time to create the sine wave animation. Because the input range of the sin function is 360 degrees (6.28

radians) we use this Distance value. The end of the sine wave now links up with the start of the sin wave producing a periodic wraparound effect.

We also want to give each vertex a colour based on its height so that the ripples can be clearly seen. We decided to make every vertex a different shade of red.

R G B A
VertexColor = (Height, 0, 0, 1)

The red colour component will be a function of the vertex height. Higher vertices will have higher intensities and vice versa. We need to calculate the red component based on a vertex height in the range of [0, 1] so we will need to map the height from the –3 to +3 range. To do this we will add half the of Y component of our scale vector to take it from [–3, +3] to [0, 6]**.** We do not want the lowest vertices to be totally black so we will add on minimum color value of 4.0 as shown below.

```
            // Calculate the color of the vertex
            float fRed = (fHeight + ( vecScale.y / 2.0f )) + 4.0f;
```

We are adding the minimum color in world space values since the color is based on the vertex height. For example, let us imagine the current vertex had a height of +1.5. The red component would be calculated as:

1.5 +3 + 4 = 8.5

If we divide this red value by the maximum vertex height taking into account we have added 4.0 we get code that looks like this:

fRed = fRed / (vecScale.y + 4.0f); // Normalize the colour value

which results in:

8.5 / 10 = 0.84

It is not a perfect linear mapping but it is easy to do and creates values relative to the height of the vertex.

We now know everything we need to know about the vertex. We have its height, its colour, and we also know its X and Z components. We use the D3DCOLOR_COLORVALUE macro to pack the four floats into a DWORD representation of the colour and we are done.

```
            *pVertex++ = CVertex( x * vecScale.x, fHeight, z * vecScale.z,
                                  D3DCOLOR_COLORVALUE( fRed, 0.0f, 0.0f, 1.0f ) );
        } // Next Column
    } // Next Row

   m_pVertexBuffer->Unlock( );
}
```

Here is the function in its entirety:

```
void CGameApp::AnimateObjects()
{
    static float Distance = 6.28f;

    ULONG       x, z;
    float       fHeight;
    CVertex    *pVertex  = NULL;
    D3DXVECTOR3 vecScale = D3DXVECTOR3( 4.0f, 30.0f, 4.0f );

    // Work out time shift
    Distance -= 5.0f * m_Timer.GetTimeElapsed();
    if (Distance < 0.0f) Distance += 6.28f; // (2*PI)

    // Lock the vertex buffer
    m_pVertexBuffer->Lock( 0, sizeof(CVertex) * (BlockWidth * BlockHeight),
                                                (void**)&pVertex, D3DLOCK_DISCARD );

    // Loop through each row
    for ( z = 0; z < BlockHeight; z++ )
    {
        // Loop through each column
        for ( x = 0; x < BlockWidth; x++ )
        {
            // Calculate height of the vertex
            float fx = ((BlockWidth / 2.0f) - x);
            float fz = ((BlockHeight / 2.0f) - z);
            float fDist = sqrtf(fx * fx + fz * fz) + Distance;
            fHeight = sinf(fDist) * (vecScale.y / 2.0f );

            // Calculate the color of the vertex
            float fRed = (fHeight + (vecScale.y/2.0f)) + 4.0f;
            fRed = fRed / (vecScale.y + 4.0f); // Normalize the colour value

            *pVertex++ = CVertex( x * vecScale.x, fHeight-50, z * vecScale.z+5,
                                  D3DCOLOR_COLORVALUE( fRed, 0.0f, 0.0f, 1.0f ) );
        } // Next Column
    } // Next Row

    m_pVertexBuffer->Unlock( );
}
```

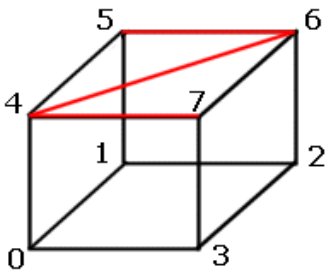
Questions and Exercises

1. Why would you want to place a vertex buffer in system memory even on a hardware-processing device?
2. Does creating a vertex buffer with the `D3DPOOL_DEFAULT` flag always place it in video memory?
3. Can we use indices to eliminate duplicate vertices under all circumstances?
4. What vertex buffer type can be rendered faster: static or dynamic?
5. Why is locking a static vertex buffer in a time critical situation a bad move?
6. What does the `D3DLOCK_DISCARD` flag do and when should it be used?
7. Can we render triangle lists using indices or are we limited to rendering only indexed triangle strips?
8. What is a degenerate triangle?
9. Should we always try to store as many vertices as possible in a vertex buffer?
10. Why is reading from AGP memory slow?
11. What is a height map?

Appendix A - Representing a Cube as an Indexed Triangle Strip

In this lesson, we learned how degenerate triangles can be used to create a continuous triangle strip even when the triangles that need to be rendered do not form a consecutive line of primitives. We used degenerate triangles to move from the end of one row up to the beginning of another row in such a way that the rows were connected by invisible triangles.

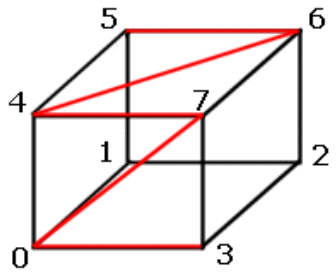
In the BuildObjects function of Lab Project 3.2 we built a cube mesh as an indexed triangle strip to be rendered when the camera is in 3rd person mode. We did this by inserting a single additional index, which created two extra triangles. Interestingly, these two triangles will not actually be degenerate. Instead they will be inward facing such that they cannot be seen from outside of the cube mesh. The following image shows the vertex positions in the cube as well as the first four indices in the list, which create the two triangles that form the top face of the cube.



Indices:
5,6,4,7

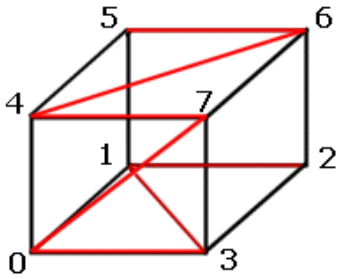
Remember that for indexed tri-strip, after the first two indices in the index array, every additional index from that point on creates a new primitive using the new index and the last two indices from the previous triangle. Also remember that unlike other primitive types, the driver expects every even numbered triangle in the list to have a clockwise winding order in view space and every odd numbered triangle in the list to have a counter-clockwise winding order in view space. In the image above, if the camera was looking down at the top face of the cube, the first triangle (0) is clockwise and the second triangle (1) is counter-clockwise so they would both be considered to be facing the camera.

Next, we added two more indices and created the next two triangles. These triangles created the front face of the cube. The first triangle has a clockwise winding order when viewed from the front and the second face is counter-clockwise as expected. If the camera were positioned in front of this cube looking directly at the front face, both of these faces would be considered to be facing the camera and would not be back face culled. As we can see in the next image, adding index 0 creates the new triangle (4,7,0) and adding index 3 creates the second triangle of the front face (7,0,3).



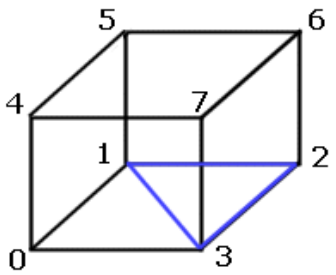
Indices:
5,6,4,7
0,3,

We now have the top and front faces represented as a single strip. Next we add an index to vertex 1 and another to vertex 2 which creates triangles (0,3,1) and (3,1,2) respectively.



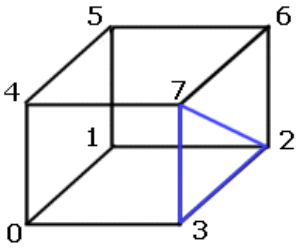
Indices:
5,6,4,7
0,3,1,2

At this point we have the top, front, and bottom faces of the cube stored as a continuous strip of triangles. This is where things get tricky. We could carry on up the back side of the cube in the same way but then there would be no way to render the left and right face without breaking our strip or generating unattractive in-between triangles. What we can do instead is use the fact that because we have just rendered an odd triangle, the next triangle must be an even triangle. The even triangle should have its vertices in clockwise order in view space when the triangle is viewed from the bottom side. We add index 3, which we can call a *degenerate index* because it is used to create triangles that will not be seen but are used as a way of moving from one vertex to another in the strip. When combined with the previous two vertices, this creates triangle (1,2,3). This is very important because the driver will be expecting a clockwise winding order for this triangle. We have just defined it using vertices that will be viewed as counter-clockwise in view space (just imagine a camera under the cube looking up at the bottom face). This triangle will be culled.



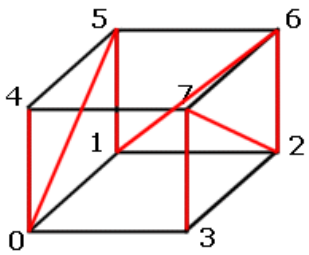
Indices:
5,6,4,7
0,3,1,2
3

Now when we add the next index (index 7) this creates a triangle on the right side of the cube (2,3,7). You might think at first that it should be visible since it is obviously clockwise, but remember that we are on an odd triangle now. The driver will consider only counter-clockwise view space winding orders visible in this case. What we have just done is have switched the clockwise/counterclockwise order of the last two triangles so that they are never seen from the outside.



Indices:
5,6,4,7
0,3,1,2
3,7

Now we have the strip where we want it. By adding the next index (to vertex two) we are back on an even triangle and we have a clockwise winding order creating triangle (3, 7, 2). Since we are back on track, we can simply add an index to vertex 6 next to create the second triangle of the left side of the cube (7,2,6) and carry on around the remaining two faces (the back and right sides). To do this we simply add the indices in the correct order as shown in the following and final image:

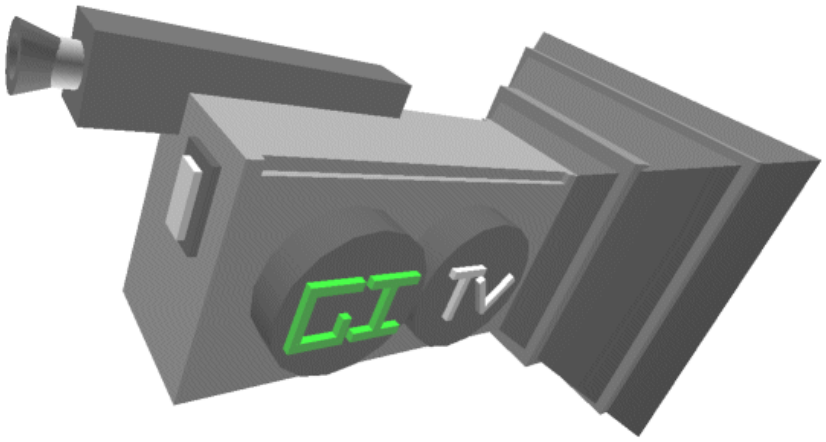


Indices:
5,6,4,7
0,3,1,2
3,7,2,6
1,5,0,4

To sum this method up, we represent the top, front, and bottom faces as a normal strip. We then insert an additional index to vertex 3, which creates two incorrectly ordered triangles. This gets us into the position where we can then represent the left, back, and right faces in the same way.

The result is one cube, eight vertices, fourteen triangles and a single call to DrawIndexedPrimitive. Despite the two additional triangles, this will be much more efficient than rendering each face with a separate call to DrawPrimitive.

Workbook Chapter Four:
Camera Systems



In this lesson we will implement and examine the code for a reusable camera system for our games. We begin with a brief introduction to the overall design goals and then proceed to examine the source code. Note that the camera system we study in this lesson is the same one used in Lab Project 3.2. We will learn how to:

- implement first person, third person and cockpit style camera perspectives
- perform local and world transformations on our camera
- move our camera around in the environment with some simple physics
- use the camera frustum to cull geometry that cannot be seen

Lab Project 4.1: Designing a Camera System

We would like to implement the three camera types so that they can be used together in a single application and can be seamlessly switched from one to the next. It would be nice for example, to be able to attach a camera to any non-player character in your game world and be able to change between first person camera mode (where you are looking through its eyes) and third person camera mode where you are following the character itself. We will implement this connectivity between cameras and meshes by implementing two new classes. The first class will be the CCamera class which has the core functionality you might expect. The second class we will implement will actually be the class we use to indirectly control the camera. This class will be called CPlayer and it has the following qualities:

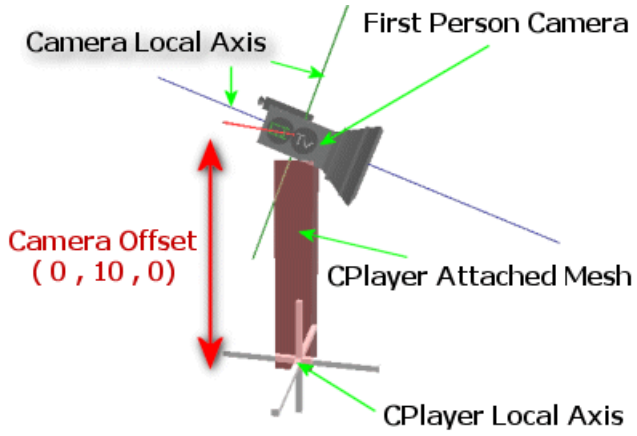
- Our application will not directly move the camera. It will attach the camera to a CPlayer object. Our application will call functions such as CPlayer::Move. This will move the player and the camera that is attached to the player depending on whether the Player has been put into first person mode, third person mode, or space craft camera mode.
- The CPlayer object can have a CObject attached to it. This is the CObject class we have been using in previous demos which basically contains a world matrix for the object and the CMesh of the object.
- If the CPlayer is in third person camera mode, the CObject has its mesh rendered, and the camera is placed at the offset from the model that we specify. Moving the CPlayer will move the CObject model. The camera is reset when the player is moved so that it follows the model. The camera remains at a distance from the mesh specified by the CPlayer's offset vector which we set with a call to CPlayer::SetCamOffset. This is totally configurable.
- In first person mode, the CMesh is not rendered. In this mode you can think of the player as being the body position of the mesh, and the camera as being at the position of the mesh's head. The CPlayer offset vector is used to offset the camera much like it is in third person mode. In our demonstration, we set the CPlayer object into first person mode at application start-up and call CPlayer::SetCamOffset with a vector of (0, 10 ,0). This places the camera 10 units vertically above the position of the CPlayer object.
- In space craft camera mode, we set the camera offset to zero (0,0,0) where the camera is situated exactly at the position of the CPlayer. This can all be configured, but seemed nice for our demonstration.
- The CPlayer object will experience gravity. This will be configurable with a call to CPlayer::SetGravity. A vector is passed by the application describing the direction and magnitude of the gravity vector. We use a gravity vector of (0,-400, 0) which seemed to work nicely for our demo. The gravity vector is applied to the CPlayer object each frame so that the CPlayer always has a downward acceleration applied to it. Obviously, if the CPlayer object is already on the terrain, then this is cancelled out, but if the CPlayer object finds itself in mid air (such as if you walk off the edge of one of the higher parts of the terrain) the CPlayer will fall to the ground. The rate at which it falls depends on the length of the gravity vector combined with how we set the maximum Y velocity (more on this later). Notice that we could send in a vector such as (0,500,0) which would actually be like gravity in reverse where the CPlayer would naturally float upwards to simulate buoyancy although it is probably not very useful to do it this way.
- We can set a camera lag value which provides a more fluid camera tracking system in third person mode. This way, when the CPlayer rotates, the camera does not rotate instantly like it is stuck on a big wooden rod. Instead, the player will rotate first with the camera catching up a fraction of a second later. We call the CPlayer::SetCameraLag function to specify in seconds the lag that we desire. Specifying a value of 0.25 for example would cause a ¼ of a second delay between the CPlayer rotating and the CCamera realigning itself with the player. The lag value is only applied to the camera in third person camera mode.
- The player object can have a friction value applied each frame which is set with a call to CPlayer::SetFriction. This allows the camera to slow to a halt gracefully instead of just coming to an abrupt stop when the forward key is released. So when we press the forward key, we apply acceleration along the velocity vector. If we had no friction, then one tap of the key would set the velocity vector and then this velocity vector would be added to the camera position every frame. This means, one tap on the key would make our CPlayer travel on forever even if the key

was released. By specifying a friction value, this value is decreased from the velocity vector each frame. If the key is being held down (and providing we set our friction correctly) the acceleration being applied by the key press will overpower the friction allowing us to accelerate to some maximum full speed. Once we release the key however, and we are no longer applying any forces, the friction value will decrement the speed in small portions until it becomes zero and our player comes to a complete stop. If the friction you specify is larger than the acceleration applied each frame, your CPlayer object will not move at all since it does not have enough forward momentum to break through the friction force. All of this will be configurable so you can tailor the system to suit your needs.

While this seems like quite a lot of stuff too implement, it is not nearly as difficult as it sounds once we understand the basic system. Let us first review how the CPlayer and CCamera classes will be used before we cover the code that actually implements their functionality.

CPlayer / CCamfirstPerson Overview

In first person mode, you can think of the CPlayer object as being the body of the player where the attached camera is its head. Moving the body of the player moves its head, but the head may also be rotated independently (looking up and down for example). So the CPlayer object and the CCamera object will need to maintain a set of local axes (Up, Look, and Right vectors). Usually, when we set the CPlayer object into first person mode, we will want to specify some offset using CPlayer::SetCamOffset to position the head in the correct position. In our application we use a vector of (0, 10, 0) so the camera is placed at a distance of 10 units above the player to emulate roughly where the position of the head might be in relation to the body. The CPlayer axes define the player's local coordinate system where the axes meet at the feet of the player. The camera local axes define the camera local coordinate system where the axes meet in the center of the players head.



The CPlayer object can be rotated about all three axes with a call to CPlayer::Rotate (X, Y, Z).

X Rotation (Pitch)

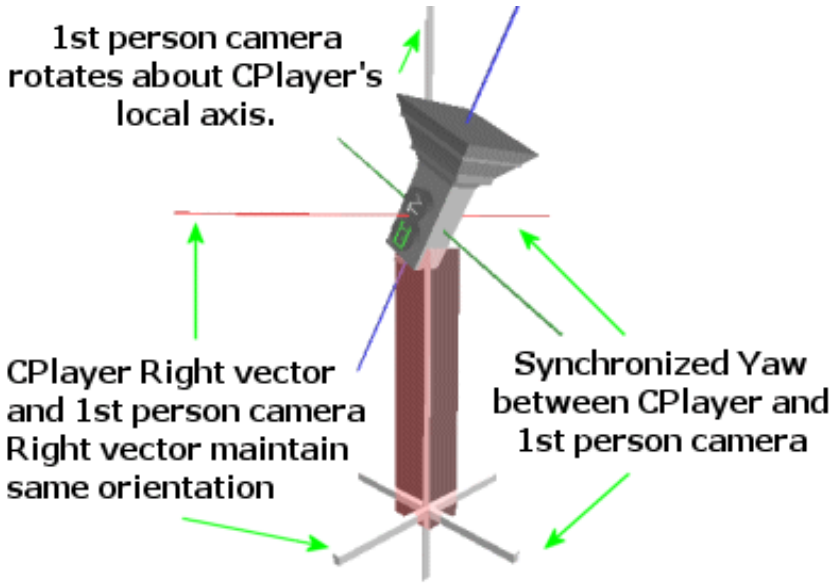
When we specify an X axis rotation, we wish to pitch the player upwards or downwards. It makes little sense for the player object to rotate about its own X axis. In real life if we wanted to look up or down we would rotate our heads back or forward, not our whole body. So, when the player is in first person camera mode, the rotation request is passed straight to the camera class by calling the CCamfirstPerson::Rotate(X,Y,Z) function. The first person camera class rotates the camera about its

own right vector (not the player right vector) allowing us to pivot the camera up and down like a head belonging to a body, as shown in the previous picture. Even though the CPlayer object does not directly pitch itself, it does retain the current pitch angle of the camera and restricts rotations past 89 degrees in both directions. This mirrors real life where your head will only look up or down so far and will not rotate completely around so that you are looking behind you. So then, in first person mode, the camera pitches by a limited amount around its own right vector.

Y Rotation (Yaw)

The Y axis rotation is the one rotation that does physically affect the orientation of the player in first person mode. This is because the Y axis rotation is like the body of the person rotating himself/herself left or right so that they are now facing in a new direction. Therefore, when a Y axis rotation is specified, both the head (the camera) and the body (the player) are rotated about the player Up vector (local Y Axis). The result is that the right vectors of both the CCamera and the CPlayer remain synchronized because they always yaw together.

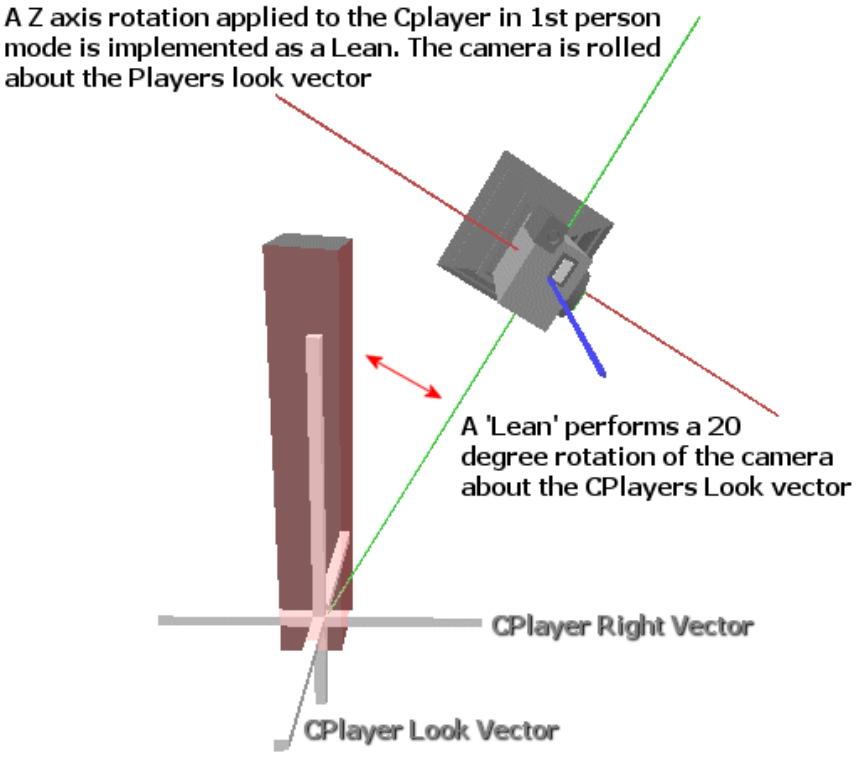
The following image shows how a Yaw works. The camera has already been pitched up so that it is rotated about its own Right vector. This is like the user rotating his head up to look at something in the sky. Now if we were to perform a Y axis rotation on the player, he would rotate about his own Up vector and forward the Y rotation request to the camera. The CCamfirstPerson class would rotate the camera about the player Up vector so that they both yaw about the same axis together.



Z Axis Rotation (Roll)

One might think that the player should ignore Z axis rotation requests when in first person camera mode. But we will actually use this request to lean the camera. This is used in many of today's games (Metal Gear Solid™, Splinter Cell™, etc.) to give the user the ability to poke his head around a corner to see what is coming without revealing his body as a target.

When the player is in first person camera mode, the CPlayer object does not do anything with the request except record the current roll angle of the camera and limit this roll angle if it is greater than 20 degrees (an angle that worked well for our purposes). After that, it sends the request to the first person camera class which rotates itself about the player Look vector as the following image shows:



The image above shows the effect of using our demo application values for camera offset. As you can see, the camera is not rotated about its own look vector, but is instead rotated about the look vector of the CPlayer object. This is basically like rotating the camera about the player's feet by 20 degrees. If you imagine the camera to be the player's head and the CPlayer coordinate system origin to be the feet, the green line which is the camera Up vector (local Y Axis) represents the attitude of the leaning body. One could argue that we are in fact dislocating the head from the body here, but remember that in first person mode there is no physical body that we are rendering. To do the same thing in third person mode would require an animation for the character such that it looks like they are peering around the corner when this request is made.

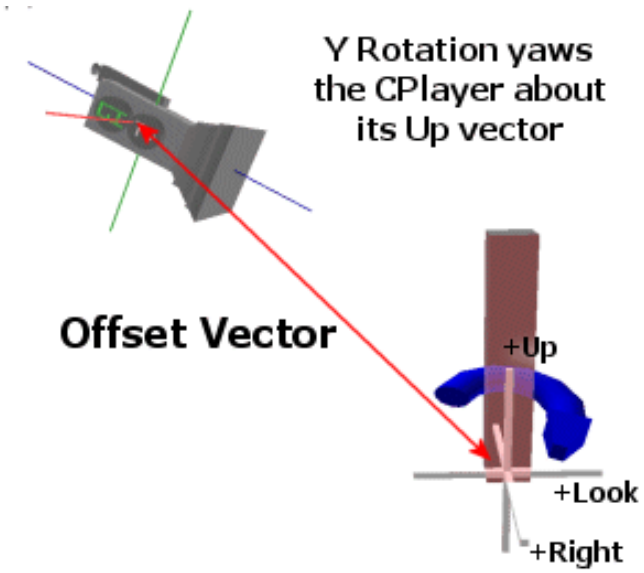
CPlayer / CCamthirdPerson Overview

The nice thing about deriving all the camera classes from a single base class is that the CPlayer does not need to know which CCamera derived class it is using. It can simply call all functions through the base class interface. Therefore, if a CCamera derived class does not want to do anything in reaction to the player being rotated about his X axis, it can simply ignore such requests in its own rotation

function. The player can still call the CCamera::Rotate function to pass on the request, but what the camera class actually does with that request is up to the camera. The CCamthirdPerson makes use of this fact to some extent. Firstly, the camera is never rotated explicitly at all by the CPlayer object (or the application). In fact it does not even implement a rotate function. When CPlayer calls the CCamthirdPerson::Rotate function, the base class implementation is called -- which does nothing at all. Furthermore, whenever the CPlayer is updated, it calls the CCamera::Update function, which is also ignored in the case of the first person and space craft camera classes. But in the case of the third person camera, this function is used to align the camera so that it looks at the player model center point.

When we set the CPlayer into third person mode, we will specify an offset vector which describes how far and in what direction to offset the camera position from the player model. We choose to place the camera behind and slightly above the player in our code but you can change this so that you are always viewing the player from the front if desired. Usually, when you place the CPlayer into third person mode you will want to attach a CObject to the CPlayer object by using the CPlayer::SetthirdPersonObject function. Unlike all other modes, when in third person mode, this object (if it has been attached) will be rendered when a request to CPlayer::Render is issued from our main render loop (in CGameApp::FrameAdvance). If we are not using third person mode, the call to CPlayer::Render does nothing.

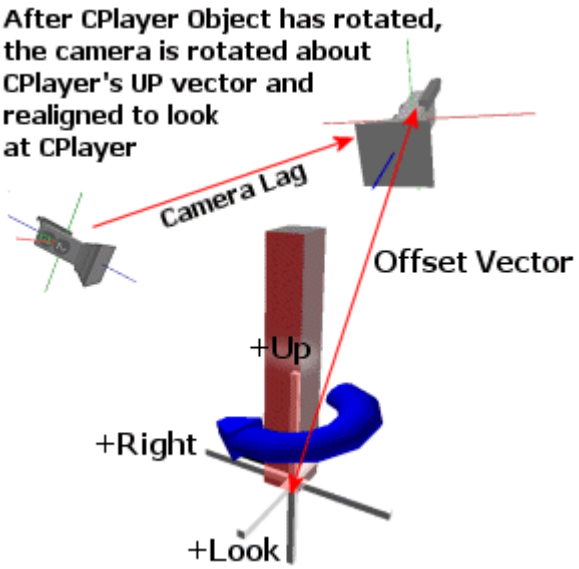
In third person mode, CPlayer ignores X and Z rotations passed into the CPlayer::Rotate(X,Y,Z) function. This is because the player can only rotate left and right. This was true in first person mode also, but we had the ability to pitch or roll the camera as the head of the player. In response to a Y rotation, the player is rotated (which also rotates its attached mesh) about his Up vector.



When the player is rotated about his Up vector, no rotation is applied to the camera initially. Later in the CPlayer::Update function (called every frame from ProcessInput) the camera offset vector is

rotated around the player Up vector by the same amount. This retrieves a point in space that is ideally where we wish the camera to rotate to in order for us to keep the relationship between the camera and player the same after the rotation. The CCamthirdPerson camera class then calculates a vector from its previous position to the new ideal position and moves along this vector. The speed at which it moves along this vector is determined by specifying the camera lag value. In our application we set a lag value of 0.25 which is ¼ second. This means the camera now drifts into its new position instead of instantly just being there to create a much more fluid feeling. So the camera will not reach its ideal position until ¼ of a second later. In the meantime it is travelling along the camera lag vector shown in the following diagram. Every frame however, the camera is made to look at the CPlayer object, so that even whilst the camera is travelling along the camera lag vector, it is always constantly updated to look at the CPlayer object.

In the following image, the red block represents the CMesh (for now, just imagine that it is a really cool character model from Unreal 2™) which belongs to the CObject attached to the CPlayer. The CPlayer can only yaw in this mode.



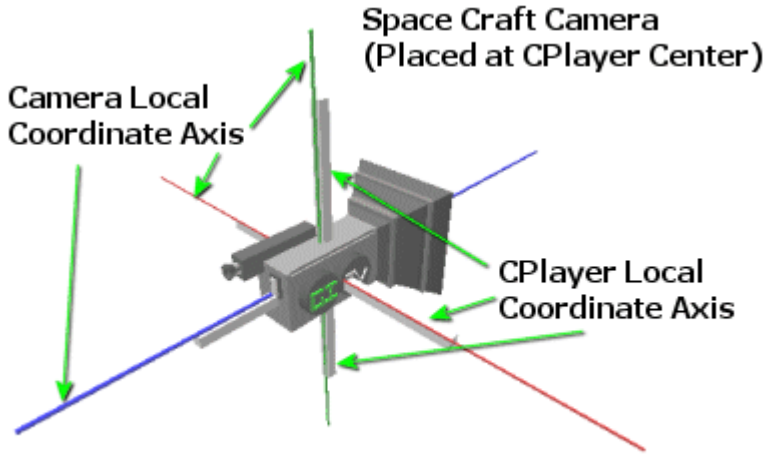
We will take a close look at the code to all the camera classes in just a bit. For now we just need to understand what we will need each camera class to accomplish.

CPlayer / CCamSpaceCraft Overview

When the player is in space craft mode, the camera used will be a CCamSpaceCraft object. This is actually the easiest camera mode to understand since it works exactly like the code we studied in the textbook. Unlike the other two modes where the CPlayer could only be rotated about the Y axis, in this mode we can rotate about all axes. Thus the space craft camera must also rotate itself about the player

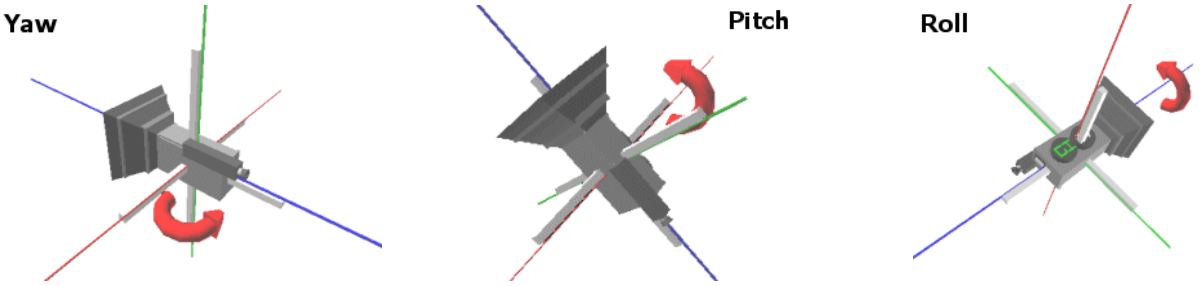
in the same way. You will probably never use a camera offset vector in space craft mode, but it is perfectly acceptable to do so. This would come in handy if your player object was attached to a space ship model when in third person mode so that their origin was positioned at the center of mass for the ship. If the user places the camera into space craft mode (to actually fly the ship) you might want the camera to be positioned high above the center of the CPlayer object to emulate looking through the window of the bridge tower. Even so, the camera and CPlayer have their rotations synchronized so that their Look, Right, and Up vectors always share the same orientations.

In our demo application, we set the camera offset to zero when the CPlayer is placed into space craft mode. This places the camera at the origin of the player's coordinate system so that both the camera position and axis vectors are an exact match of the CPlayer position and axis vectors:



No offset between CCamera and CPlayer objects

In this mode, X, Y, and Z rotations are allowed. This rotates the CPlayer about its own local axes allowing us to perform pitch, yaw and roll whilst the camera is also pitched, yawed and rolled about the same axes. Because we use a camera offset of 0 for this mode, this has the effect of rotating the camera about its own local axes.



CGameApp::SetupGameState

If we revisit the CGameApp::SetupGameState function from Lab Project 3.2, we will see that this is where we initially place the camera (indirectly through the CPlayer object) into first person mode. We specify an offset vector of (0, 10, 0) which places the camera 10 units above the position of the CPlayer object. We set the camera lag to zero since this is not used by the first person camera, and we also set friction to 250 units per second. We pass a gravity vector that pushes vertically down on the Y axis at 400 units per second.

```
void CGameApp::SetupGameState()
{
    // Generate an identity matrix
    D3DXMatrixIdentity( &m_mtxIdentity );

    // App is active
    m_bActive = true;

    // Setup the players camera, and extract the pointer.
    // This pointer will only ever become invalid on subsequent
    // calls to CPlayer::SetCameraMode and on player destruction.
    m_Player.SetCameraMode( CCamera::MODE_FPS );
    m_pCamera = m_Player.GetCamera();

    // Setup our player's default details
    m_Player.SetFriction( 250.0f ); // Per Second
    m_Player.SetGravity( D3DXVECTOR3( 0, -400.0f, 0 ) );
    m_Player.SetMaxVelocityXZ( 125.0f );
    m_Player.SetMaxVelocityY ( 400.0f );
    m_Player.SetCamOffset( D3DXVECTOR3( 0.0f, 10.0f, 0.0f ) );
    m_Player.SetCamLag( 0.0f );

    // Set up the players collision volume info
    VOLUME_INFO Volume;
    Volume.Min = D3DXVECTOR3( -3, -10, -3 );
    Volume.Max = D3DXVECTOR3(  3,  10,  3 );
    m_Player.SetVolumeInfo( Volume );

    // Setup our cameras view details
    m_pCamera->SetFOV( 160.0f );
    m_pCamera->SetViewport(m_nViewX,m_nViewY,m_nViewWidth,m_nViewHeight,1.01f,5000.0f);

    // Set the camera volume info (matches player volume)
    m_pCamera->SetVolumeInfo( Volume );

    // Add the update callbacks required
    m_Player.AddPlayerCallback( CTerrain::UpdatePlayer, (LPVOID)&m_Terrain );
    m_Player.AddCameraCallback( CTerrain::UpdateCamera, (LPVOID)&m_Terrain );

    // Lets give a small initial rotation and set initial position
    m_Player.SetPosition( D3DXVECTOR3( 430.0f, 400.0f, 330.0f ) );
    m_Player.Rotate( 25, 45, 0 );
}
```

The CPlayer object creates the CCamera derived first person camera object and we retrieve a pointer to it. We set the friction, gravity, camera offset, camera lag, and the maximum velocity in both the XZ

direction, and the Y direction. The XZ maximum velocity is the maximum amount of speed our camera can travel (horizontally) across the terrain. 125 units per second is the ceiling for this demo. We apply a much faster downwards maximum velocity which makes sense -- we should fall much faster than we can walk. We also set a volume for both the camera and the player object to be used for collision detection against the terrain. These volume functions will be covered later on.

Finally, we call the CPlayer::AddPlayerCallback and CPlayer::AddCameraCallback functions to add a function pointer to both of the CPlayer call-back arrays. This will be covered in more detail later. At a high level, when we update the CPlayer object each frame (by calling CPlayer::Update) the CPlayer -- and its attached camera -- will be moved to a new position. After this has happened, the CPlayer object has an array of function pointers that it can call so that external objects can agree to the changes in position, or possibly update the position of the CPlayer or the camera object if it is not valid. The CTerrain class has a function called UpdatePlayer. When the player updates their position based on user input (and gravity) it calls the CTerrain::UpdatePlayer function. This function will check the new position of the CPlayer object and if it has fallen into or through the terrain it will correct the CPlayer position so that it is correctly placed on top of the terrain. The same thing happens when the CPlayer moves the camera. It calls the CTerrain::UpdateCamera function to give the terrain a chance to update the camera position so that it does not get embedded inside the terrain. Although we only add a single callback for the camera and the player, it is possible to add many more. This may be useful if we had a terrain with a few scenery meshes (like trees for example). Each scenery mesh could add a call-back function to the CPlayer array so that it can check that the camera or player has not collided with it when the camera or player is updated. Just to be clear, this most definitely is *not* the type of collision detection system we will build later in the curriculum, but it does serve our purposes for now due to its simplicity and the relatively small scenes we are using.

We set the player parameters according to the camera mode we intend to use. The following code snippet is from the CGameApp::DisplayWndProc function in Lab Project 3.2. It is executed in response to the user selecting third person camera mode from the menu.

```
case ID_CAMERAMODE_THIRDPERSON:
 // Set camera mode to third person style
 ::CheckMenuRadioItem( m_hMenu, ID_CAMERAMODE_FPS, ID_CAMERAMODE_THIRDPERSON,
                               ID_CAMERAMODE_THIRDPERSON, MF_BYCOMMAND );

 // Setup Player details
 m_Player.SetFriction        ( 250.0f ); // Per Second
 m_Player.SetGravity         ( D3DXVECTOR3( 0, -400.0f, 0 ) );
 m_Player.SetMaxVelocityXZ   ( 125.0f );
 m_Player.SetMaxVelocityY    ( 400.0f );
 m_Player.SetCamOffset       ( D3DXVECTOR3( 0.0f, 40.0f, -60.0f ) );
 m_Player.SetCamLag          ( 0.25f ); // 1/4 second camera lag

 // Switch camera mode
 m_Player.SetCameraMode      ( CCamera::MODE_THIRDPERSON );
 m_pCamera = m_Player.GetCamera();
 break;
```

When we place the player into third person mode, we set a camera offset vector that is initially 40 units above the player mesh and 60 units behind it. We also set the camera lag to ¼ of a second.

The next snippet of code from the same function shows how the settings are changed again to accommodate the camera being put into space craft mode.

```
case ID_CAMERAMODE_SPACECRAFT:
 // Set camera mode to SPACECRAFT style
 ::CheckMenuRadioItem( m_hMenu, ID_CAMERAMODE_FPS, ID_CAMERAMODE_THIRDPERSON,
                                ID_CAMERAMODE_SPACECRAFT, MF_BYCOMMAND );

 // Setup player details
 m_Player.SetFriction        ( 125.0f ); // Per Second
 m_Player.SetGravity         ( D3DXVECTOR3( 0, 0, 0 ) );
 m_Player.SetMaxVelocityXZ   ( 400.0f );
 m_Player.SetMaxVelocityY    ( 400.0f );
 m_Player.SetCamOffset       ( D3DXVECTOR3( 0.0f, 0.0f, 0.0f ) );
 m_Player.SetCamLag          ( 0.0f ); // No camera lag

 // Switch camera mode
 m_Player.SetCameraMode      ( CCamera::MODE_SPACECRAFT );
 m_pCamera = m_Player.GetCamera();
 break;
```

In space craft mode, we zero out the gravity vector so that no gravity is applied. This allows us to hover in the air and fly about the terrain.

Player Controls

CGameApp::ProcessInput

In Lab Project 3.2, a function called CGameApp::ProcessInput is called every frame from the CGameApp::FrameAdvance function. This function is responsible for reading the current state of the keyboard and moving/rotating the player object in response to the keys that are currently pressed. It is also responsible for calling the CPlayer::Update function which applies any movement, rotation, friction, and gravity to the velocity vector of the CPlayer object. This CPlayer object will also take care of updating its currently attached camera so that it is positioned correctly. This means that we do not have to explicitly move the camera from our CGameApp class. Once the CPlayer object has been updated, we call the CTerrain::UpdatePlayer function. This function makes sure that the player -- or the camera -- is not embedded in the terrain and correctly positions the camera/player so that it is on the terrain at the correct height. Without this function, the gravity vector would allow our player/camera to fall right through the terrain.

At this point, the player and camera variables have been updated to represent the new player and camera positions. All that is left to do is to instruct the camera object to build a new view matrix based on its current variables and send it to the device.

We will now look at the ProcessInput function a few lines at a time. It is called every frame from the CGameApp::FrameAdvance function prior to rendering the scene. When this function returns, the camera is in the correct position and the correct view matrix has been set.

```
void CGameApp::ProcessInput( )
{
    static UCHAR pKeyBuffer[ 256 ];
    ULONG        Direction = 0;
    POINT        CursorPos;
    float        X = 0.0f, Y = 0.0f;

    if ( !GetKeyboardState( pKeyBuffer ) ) return;
```

First we call the Win32 function GetKeyboardState and pass the address of a 256 element unsigned char array. This function will record the state of all 256 virtual keys into the array which we can then use to check whether a particular key is pressed. The application will use the Virtual Key Code defines as indices into the array to check the state of a particular key. For example, the virtual key code VK_UP holds the index of the byte in the array that has the state information for the UP cursor key. If the high bit of this byte is set, the key is pressed. So we can check the state of the UP cursor key with the following code.

```
if (pKeyBuffer[ VK_UP ] & 0xF0 )  { //Key is Pressed }
```

As you can see, we use the value 0xF0 to mask the high bit.

> **Note:** The status changes as a thread removes keyboard messages from its message queue. The status does not change as keyboard messages are posted to the thread's message queue, nor does it change as keyboard messages are posted to or retrieved from the message queues of other threads.

At this point we have the state information for all keys in our array. We now check the keys that are pressed and respond accordingly:

```
    if ( pKeyBuffer[ VK_UP    ] & 0xF0 )  Direction |= CPlayer::DIR_FORWARD;
    if ( pKeyBuffer[ VK_DOWN  ] & 0xF0 )  Direction |= CPlayer::DIR_BACKWARD;
    if ( pKeyBuffer[ VK_LEFT  ] & 0xF0 )  Direction |= CPlayer::DIR_LEFT;
    if ( pKeyBuffer[ VK_RIGHT ] & 0xF0 )  Direction |= CPlayer::DIR_RIGHT;
    if ( pKeyBuffer[ VK_PRIOR ] & 0xF0 )  Direction |= CPlayer::DIR_UP;
    if ( pKeyBuffer[ VK_NEXT  ] & 0xF0 )  Direction |= CPlayer::DIR_DOWN;
```

The Direction variable is a DWORD that can have several bits set indicating whether it should move backwards, forwards, left, right, up or down. To make things easier (so that we do not have to remember which bits mean what) we can use the DIRECTION enumerated type which is part of the CPlayer namespace. This is defined in CPlayer.h as shown below.

```
enum DIRECTION
{
        DIR_FORWARD      = 1,
        DIR_BACKWARD     = 2,
        DIR_LEFT         = 4,
        DIR_RIGHT        = 8,
```

```
        DIR_UP           = 16,
        DIR_DOWN         = 32,
        DIR_FORCE_32BIT  = 0x7FFFFFFF
};
```

We can bitwise OR a combination of these flags into the Direction DWORD. This will be passed into the CPlayer::Move function which extracts the bits from the DWORD and moves the camera in the specified directions.

The next section of code is only executed if the window currently has capture of the mouse. The application window captures the mouse when the left mouse button is pressed and releases capture when the left button is released. This is done in the CGameApp::DisplayWndProc function shown below. Notice that when we capture the mouse in response to a WM_LBUTTONDOWN function, we also record the position of the mouse in the CGameApp member variable m_OldCursorPos.

```
//Code Snippet from the CGameApp::DisplayWndProc function (Message Processing Function)
case WM_LBUTTONDOWN:
            SetCapture( m_hWnd );
            GetCursorPos( &m_OldCursorPos );
            break;

case WM_LBUTTONUP:
            ReleaseCapture( );
            break;
```

If capture is set in the CGameApp::ProcessInput function, then the left mouse is button currently being held down. This is important for us to know, because if this is the case, we want movement of the mouse to actually rotate the player in our scene. When the capture is not set, we want to allow the user to move the mouse cursor over the application window. The next section of code measures the offset from the previous cursor position (which was initially recorded when the mouse button was first pressed in DisplayWndProc) to the current cursor position. It divides these X and Y offsets by 3 to turn the cursor offset into an X:Y offset that it is a more suitable to be used to rotate our player object. If you wish to change this setting, the less you divide by, the faster the camera will rotate. Admittedly, this is not the most robust input system you can conceive and is certainly not optimal, but it does serve as a means for getting user input into our application. You should really use DirectInput to manage your user input in performance critical code. Although DirectInput is beyond the scope of this course, we will be looking at it in some detail later on in the course series.

```
    if ( GetCapture() == m_hWnd )
    {
        SetCursor( NULL );
        GetCursorPos( &CursorPos );
        X = (float)(CursorPos.x - m_OldCursorPos.x) / 3.0f;
        Y = (float)(CursorPos.y - m_OldCursorPos.y) / 3.0f;
        SetCursorPos( m_OldCursorPos.x, m_OldCursorPos.y );
    }
```

The above code checks whether the left button is currently down (i.e. we have capture) and sets the mouse cursor to NULL. This removes the arrow cursor from the screen and stops it from being displayed in the frame. It then gets the current position of the cursor in screen coordinates. It subtracts the old position from the new position so that we have the amount that we have moved in both the X and Y dimensions. Finally, we set the cursor position back to the previous position so that the (now invisible) mouse cursor never hits the side of the screen. If we did not do this, every time the cursor reached the extents of the screen, we lose the ability to rotate further. We obviously want the user to be able to continue yawing as long as they wish. By setting it back, we are in effect creating a treadmill concept: we record the movement, reset, and go to the next frame where the process starts all over again. The X and Y values that result from being divided by 3 will be used as degree values passed into the CPlayer::Rotate function.

At this point, we have a DWORD containing a bit set for each direction we wish to move. We also have two values (X and Y) that contain the rotation angle in degrees. The axis by which we rotate is dependant on the mode the CPlayer object is currently in: first person, third person or space craft. The following snippet of code checks how we need to rotate or move the camera.

```
    // Update if we have moved
    if ( Direction > 0 || X != 0.0f || Y != 0.0f )
    {
        // Rotate our camera?
        if ( X || Y )
        {
            // Are they holding the right mouse button ?
            if ( pKeyBuffer[ VK_RBUTTON ] & 0xF0 )
                m_Player.Rotate( Y, 0.0f, -X );
            else
                m_Player.Rotate( Y, X, 0.0f );

        } // End if any rotation
```

The above code is the core interaction between the user and the CPlayer object (and therefore indirectly, the camera). First we check if X or Y is set to some value other than zero. If so, it means that the mouse has been moved with the left button down and the user is requesting player rotation. We call the CPlayer::Rotate function in response. It accepts X, Y, and Z values describing the angles to rotate around the relative axis of rotation. If the right button is also pressed, we use the Y value (calculated from vertical mouse movements) to rotate the player about its X axis (pitch). We use the X value (calculated from horizontal mouse movement) to roll the camera.

You can roll the camera/player left or right by holding down both mouse buttons and performing horizontal mouse movements. If the CPlayer is in first person camera mode, this causes the camera to rotate about the player's center point and emulates a 'lean around a corner' manoeuvre. If the CPlayer is in space craft mode then this will rotate the camera about its local Z axis, allowing you spin upside down. In the third person camera mode, the third parameter passed into this function is ignored, since the camera cannot roll. This will all become much clearer when we look at the CPlayer and CCamera source code in a moment. If the Right mouse button is not held down (just the left is), then a typical rotation occurs. The X value is still used for pitch but the Y value is used for yaw. If the CPlayer is in first person mode, this causes the camera to rotate about the world Y axis. In space craft mode, this

allows the camera to rotate about its own local Y axis (Up vector). Finally, if we are in third person mode, the camera rotates around the player (the mesh) at a radius specified by the offset vector when the class was initialised. The offset was set with a call to CPlayer::SetCamOffset.

At this point any rotations that needed to be applied will have been applied. The player will have been rotated and the camera attached to the player will be set as well. Next we need to check whether any movement (translation) is required. If any of the bits have been set in our local scope Direction variable, it means that at least some keys were pressed. If this is the case, we call the CPlayer::Move function to move the player/camera in the world. We pass in the Direction DWORD so that the CPlayer object knows what keys were pressed, and we also pass an acceleration value (500 units per second in this case). The final parameter specifies whether you wish this acceleration to be applied to the player's velocity vector or simply applied as an absolute movement:

```
        // Any Movement ?
        if ( Direction )
        {
            // Move our player (Force applied must be greater than total friction)
            m_Player.Move( Direction, 500.0f * m_Timer.GetTimeElapsed(), true );

        } // End if any movement

    } // End if some movement occurred
```

If we pass in true as the third parameter (this is the default) then the distance is added to the velocity vector calculated in the CPlayer::Move function. Since we use friction in our demo, this value is not necessarily the distance you will move.

If we pass in false as the third parameter, then the player and camera are physically moved 500 (in our example) units along the direction vector calculated by the CPlayer::Move function. This is an absolute translation from its current position along the direction vector by the specified amount.

The actual processing of the CPlayer object is in the CPlayer::Update function. This is where the friction and gravity are applied and where the player and camera are moved along the velocity vector. The Update function then loops through its internal list of camera call-back functions and player call-back functions calling each one in turn. These call-backs may modify the position of the CPlayer or the Camera if the movement was illegal.

The following code is called every frame to ensure that all gravity, friction, and velocity can be applied in a time relative fashion.

```
    // Update our camera (updates velocity etc)
    m_Player.Update( m_Timer.GetTimeElapsed() );
```

At this point, the camera and player have been moved and the terrain class has sorted out any possible collisions so that the camera and player are in legal positions on the terrain. All that is left to do now is to instruct the attached camera class to build a view matrix based on its internal variables and send it off to the device.

```
    // Update the device matrix
    m_pCamera->UpdateRenderView( m_pD3DDevice );
}
```

If you look at the complete code for the CGameApp::ProcessInput function you will see that it is actually very straightforward:

```
void CGameApp::ProcessInput( )
{
    static UCHAR pKeyBuffer[ 256 ];
    ULONG        Direction = 0;
    POINT        CursorPos;
    float        X = 0.0f, Y = 0.0f;

    // Retrieve keyboard state
    if ( !GetKeyboardState( pKeyBuffer ) ) return;

    // Check the relevant keys
    if ( pKeyBuffer[ VK_UP    ] & 0xF0 ) Direction |= CPlayer::DIR_FORWARD;
    if ( pKeyBuffer[ VK_DOWN  ] & 0xF0 ) Direction |= CPlayer::DIR_BACKWARD;
    if ( pKeyBuffer[ VK_LEFT  ] & 0xF0 ) Direction |= CPlayer::DIR_LEFT;
    if ( pKeyBuffer[ VK_RIGHT ] & 0xF0 ) Direction |= CPlayer::DIR_RIGHT;
    if ( pKeyBuffer[ VK_PRIOR ] & 0xF0 ) Direction |= CPlayer::DIR_UP;
    if ( pKeyBuffer[ VK_NEXT  ] & 0xF0 ) Direction |= CPlayer::DIR_DOWN;

    // Now process the mouse (if the button is pressed)
    if ( GetCapture() == m_hWnd )
    {
        // Hide the mouse pointer
        SetCursor( NULL );

        // Retrieve the cursor position
        GetCursorPos( &CursorPos );

        // Calculate mouse rotational values
        X = (float)(CursorPos.x - m_OldCursorPos.x) / 3.0f;
        Y = (float)(CursorPos.y - m_OldCursorPos.y) / 3.0f;

        // Reset our cursor position so we can keep going forever :)
        SetCursorPos( m_OldCursorPos.x, m_OldCursorPos.y );

    } // End if Captured

    // Update if we have moved
    if ( Direction > 0 || X != 0.0f || Y != 0.0f )
    {
        // Rotate our camera
        if ( X || Y )
        {
            // Are they holding the right mouse button ?
            if ( pKeyBuffer[ VK_RBUTTON ] & 0xF0 )
                m_Player.Rotate( Y, 0.0f, -X );
            else
                m_Player.Rotate( Y, X, 0.0f );

        } // End if any rotation
```

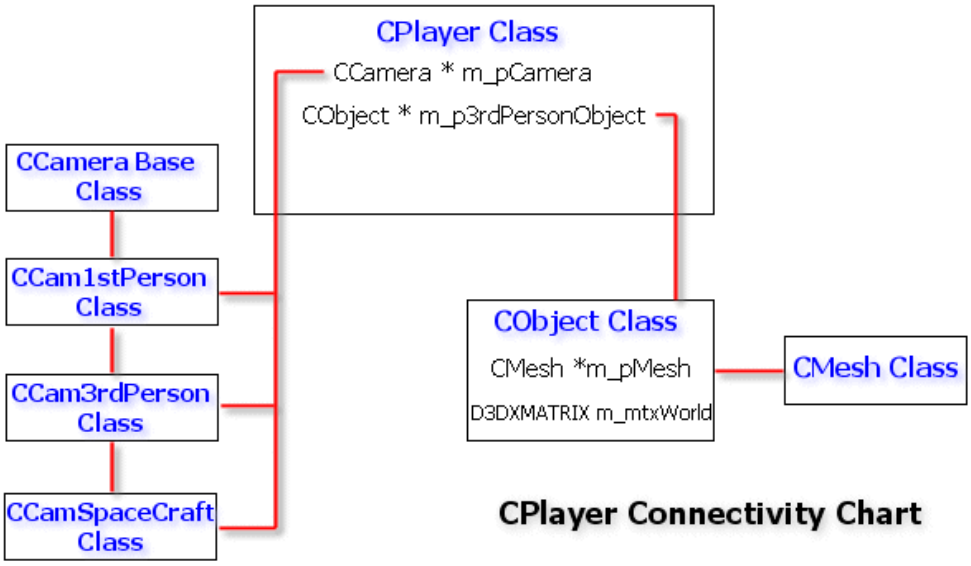
```
        // Any Movement ?
        if ( Direction )
        {
            // Move our player (Force applied must be greater than total friction)
            m_Player.Move( Direction, 500.0f * m_Timer.GetTimeElapsed(), true );

        } // End if any movement

    } // End if camera moved

    // Update our camera (updates velocity etc)
    m_Player.Update( m_Timer.GetTimeElapsed() );

    // Update the device matrix
    m_pCamera->UpdateRenderView( m_pD3DDevice );
}
```

And that is literally all there is to using the camera class. If we look back at the CGameApp::FrameAdvance function in Lab Project 3.2 we will see that we also call the CPlayer::Render method. This only renders the CPlayer object's attached mesh if the camera is in third person mode. Otherwise this function call does nothing. Below we see the section of interest from the CGameApp::FrameAdvance function:

```
    // Begin Scene Rendering
    m_pD3DDevice->BeginScene();

    // Reset our world matrix
    m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_mtxIdentity );

    // Render our terrain objects
    m_Terrain.Render( );

    // Request our player render itself
    m_Player.Render( m_pD3DDevice );

    // End Scene Rendering
    m_pD3DDevice->EndScene();
```

So let us briefly review the connectivity information between our camera, our player, and the mesh that gets rendered in third person mode. The CPlayer class ties everything together. It has a pointer to a CCamera derived class that it manages. This can be a pointer to a CCamfirstPerson, a CCamthirdPerson, or a CCamSpaceCraft class. The CPlayer object automatically destroys and recreates the relevant camera when we set it to a different mode with a call to CPlayer::SetCameraMode.

We can attach a CObject to the CPlayer class so that its mesh is rendered when we are in third person mode. The CPlayer class also manages alterations to the CObject world matrix. The CObject has a CMesh attached to it just like in previous demo applications -- where the CObject is basically just a container for a world matrix and a CMesh.



CPlayer Connectivity Chart

Coding the Camera System

The CPlayer Class

```
class CPlayer
{
public:
    //-----------------------------------------------------------------------------
    // Enumerations
    //-----------------------------------------------------------------------------
    enum DIRECTION {
        DIR_FORWARD     = 1,
        DIR_BACKWARD    = 2,
        DIR_LEFT        = 4,
        DIR_RIGHT       = 8,
        DIR_UP          = 16,
        DIR_DOWN        = 32,

        DIR_FORCE_32BIT = 0x7FFFFFFF
    };

    //-----------------------------------------------------------------------------
    // Constructors & Destructors for This Class.
    //-----------------------------------------------------------------------------
             CPlayer();
    virtual ~CPlayer();

    //-----------------------------------------------------------------------------
    // Public Functions for This Class.
    //-----------------------------------------------------------------------------
```

```
    bool      SetCameraMode      ( ULONG Mode );
    void      Update             ( float TimeScale );

    void      AddPlayerCallback    ( UPDATEPLAYER pFunc,  LPVOID pContext );
    void      AddCameraCallback    ( UPDATECAMERA pFunc,  LPVOID pContext );
    void      RemovePlayerCallback ( UPDATEPLAYER pFunc,  LPVOID pContext );
    void      RemoveCameraCallback ( UPDATECAMERA pFunc,  LPVOID pContext );

    void      SetthirdPersonObject ( CObject * pObject ){ m_pthirdPersonObject = pObject; }
    void      SetFriction        ( float Friction )     { m_fFriction = Friction; }
    void      SetGravity         ( const D3DXVECTOR3& Gravity ) { m_vecGravity = Gravity; }
    void      SetMaxVelocityXZ   ( float MaxVelocity )  { m_fMaxVelocityXZ = MaxVelocity; }
    void      SetMaxVelocityY    ( float MaxVelocity )  { m_fMaxVelocityY = MaxVelocity; }
    void      SetVelocity     ( const D3DXVECTOR3& Velocity ) { m_vecVelocity = Velocity; }
    void      SetCamLag          ( float CamLag )             { m_fCameraLag = CamLag; }
    void      SetCamOffset       ( const D3DXVECTOR3& Offset );
    void                SetVolumeInfo      ( const VOLUME_INFO& Volume );
    const VOLUME_INFO&  GetVolumeInfo      ( ) const;

    CCamera           * GetCamera       ( ) const     { return m_pCamera; }
    const D3DXVECTOR3 & GetVelocity     ( ) const     { return m_vecVelocity; }
    const D3DXVECTOR3 & GetCamOffset    ( ) const     { return m_vecCamOffset; }

    const D3DXVECTOR3 & GetPosition     ( ) const     { return m_vecPos; }
    const D3DXVECTOR3 & GetLook         ( ) const     { return m_vecLook; }
    const D3DXVECTOR3 & GetUp           ( ) const     { return m_vecUp; }
    const D3DXVECTOR3 & GetRight        ( ) const     { return m_vecRight; }

    float     GetYaw          ( ) const { return m_fYaw; }
    float     GetPitch        ( ) const { return m_fPitch; }
    float     GetRoll         ( ) const { return m_fRoll; }

    void      SetPosition( const D3DXVECTOR3& Position )
                          { Move( Position - m_vecPos, false ); }

    void      Move            ( ULONG Direction, float Distance, bool Velocity = false );
    void      Move            ( const D3DXVECTOR3& vecShift, bool Velocity = false );
    void      Rotate          ( float x, float y, float z );
    void      Render          ( LPDIRECT3DDEVICE9 pDevice );

private:
    //--------------------------------------------------------------------------
    // Private Variables for This Class.
    //--------------------------------------------------------------------------
    CCamera       * m_pCamera;              // Our current camera object
    CObject       * m_pthirdPersonObject;   // Object to be displayed in third person mode
    VOLUME_INFO     m_Volume;               // Stores information about players collision
                                            // volume
    ULONG           m_CameraMode;           // Stored camera mode

    // Players position and orientation values
    D3DXVECTOR3     m_vecPos;               // Player Position
    D3DXVECTOR3     m_vecUp;                // Player Up Vector
    D3DXVECTOR3     m_vecRight;             // Player Right Vector
    D3DXVECTOR3     m_vecLook;              // Player Look Vector
    D3DXVECTOR3     m_vecCamOffset;         // Camera offset
    float           m_fPitch;               // Player pitch
    float           m_fRoll;                // Player roll
    float           m_fYaw;                 // Player yaw
```

```
    // Force / Player Update Variables
    D3DXVECTOR3     m_vecVelocity;          // Movement velocity vector
    D3DXVECTOR3     m_vecGravity;           // Gravity vector
    float           m_fMaxVelocityXZ;       // Maximum camera velocity on XZ plane
    float           m_fMaxVelocityY;        // Maximum camera velocity on Y Axis
    float           m_fFriction;       // The amount of friction causing the camera to slow
    float           m_fCameraLag;      // Amount of camera lag in seconds (0 to disable)

    // Stored collision callbacks
    CALLBACK_FUNC   m_pUpdatePlayer[255];   // Array of 'UpdatePlayer' callbacks
    CALLBACK_FUNC   m_pUpdateCamera[255];   // Array of 'UpdateCamera' callbacks
    USHORT          m_nUpdatePlayerCount;   // Number of 'UpdatePlayer' callbacks stored
    USHORT          m_nUpdateCameraCount;   // Number of 'UpdateCamera' callbacks stored
};
```

If the class seems complex at first, just note that most of these are simple functions to set and retrieve member variables. Only a handful of functions will need to have their function bodies implemented in CPlayer.cpp. Also note that the CPlayer::SetVolumeInfo function takes as its parameter a VOLUME_INFO structure; this too is in CPlayer.h and is shown next:

```
typedef struct _VOLUME_INFO
{
    D3DXVECTOR3 Min;
    D3DXVECTOR3 Max;
} VOLUME_INFO;
```

This structure is used to represent a bounding volume by specifying two vectors that describe the minimum and maximum extents of the volume. We use it in our application to represent an axis aligned bounding box (AABB) around the player object. It is used by the CTerrain class to check for collisions with the terrain. We will discuss this in more detail later in the lesson. First we will look at the member variables that are managed by this class. They are listed below with a description of their purpose.

CCamera *m_pCamera
This is a pointer to a CCamera derived class. The CPlayer class will automatically create the appropriate camera (and then destroy the previous one) when the application requests that the CPlayer change camera modes. This is done with a call to CPlayer::SetCameraMode. This pointer is initialized to NULL.

CObject *m_pthirdPersonObject
This pointer is initialized to NULL but can point at a CObject containing the CMesh that you would like to have rendered when the CPlayer is in third person mode. If you have no intention of using third person mode, then you do not need to set this pointer. This pointer is assigned to a CObject using the CPlayer::SetthirdPersonObject function, whose body is shown above.

VOLUME_INFO m_Volume
We use this to set up a bounding box around the CPlayer object. It will be used for collision detection by the CTerrain class in this demo. We interpret the bounding volume as a bounding box, but the

VOLUME_INFO min and max vectors could be used to represent other bounding volumes such as cylinders, spheres or ellipsoids.

ULONG m_CameraMode
This contains the mode that the Player object is currently in (first person, third person, or space craft).

D3DXVECTOR3 m_vecPos
This vector stores the position of the CPlayer object in the 3D world. We can make the CPlayer object move through the world by updating this vector.

D3DXVECTOR3 m_vecUp, m_vecRight, m_vecLook
These three vectors describe the orientation of the local coordinate system axes. We can rotate the CPlayer object by rotating these vectors.

D3DXVECTOR3 m_vecCamOffset
This vector describes a camera offset from the CPlayer object.

float m_fPitch, m_fRoll, m_fYaw
These variables maintain the current rotation values in degrees applied to the CPlayer class. For example, the m_fYaw contains the current angle at which the CPlayer has been rotated about its Up vector. This allows us to apply the rotation to the attached CObject world matrix in third person mode. The m_fRoll variable contains the angle that we are currently leaning in first person mode. This is used to check that we have not tried to exceed our maximum lean angle. Likewise, m_fPitch is used in first person camera mode to maintain the current angle that the camera is pitched up or down. The CPlayer class checks this value before applying a rotation to the attached camera such that it is not rotated more than 89 degrees in either direction (up or down).

D3DXVECTOR3 m_vecVelocity
This vector is used to maintain the player direction and speed.

D3DXVECTOR3 m_vecGravity
This is the gravity vector. It will be combined with the velocity vector every frame. In space ship mode, our application sets this to zero so that we can fly the camera about in the sky without falling to the ground. In first and third person camera modes we use a gravity vector of (0,-400, 0) to apply a constant downward acceleration of 400 units. Feel free to experiment with any of these values; 400 just happened to work nicely for our demonstration.

float m_fFriction
This value contains our friction coefficient. It will be applied to the velocity vector each frame. This is used to generate a friction vector that is subtracted from the velocity vector to slow the player down. The friction vector is generated by creating a unit length version of the velocity vector, inverting it so that it faces in the opposite direction, and then scaling it by the friction value. You can also consider this to be a drag coefficient if you prefer.

float m_fCameraLag
This variable is set by calling CPlayer::SetCamLag. It is passed on to the CCamthirdPerson::Update function to control a delay (in seconds) that should be applied to the camera when rotating into a new position behind the player.

float m_fMaxVelocityXZ
This is used to set a maximum speed limit in the XZ plane that the CPlayer can move in a single frame. It is specified in world units per second.

float m_fMaxVelocityY
This is used to set a maximum speed limit that the CPlayer can move upwards or downwards in a single frame. In the first person and third person modes, we set this a fair bit higher than the MaxVelocityXZ variable, because we will want the CPlayer to fall from the sky (if you walk off a mountain edge) much faster than the CPlayer can physically walk in the XY plane. In Space Craft mode however, we set these last two values equally as this mode does not have gravity applied in our demo. Of course, you can choose to apply gravity to the space craft mode so that the spaceship slowly falls from the sky when you are not travelling upwards or if you wanted to model a more random hover pattern where the craft slowly bobs up and down. Of course, you would need to account for the upwards velocity in this latter case.

CALLBACK_FUNC m_pUpdatePlayer[255];
CALLBACK_FUNC m_pUpdateCamera[255];
These members are two arrays that can be used to hold CALLBACK_FUNC structures. The CALLBACK_FUNC structure is defined in CPlayer.h.

```
typedef struct _CALLBACK_FUNC
{
    LPVOID  pFunction;          // Function Pointer
    LPVOID  pContext;           // Context to pass to the function
} CALLBACK_FUNC;
```

When the application calls CPlayer::Update to update the position of the CPlayer and its attached camera, it is possible that the player (or its camera) can become embedded in the terrain. These call-back functions allow the terrain a chance to handle collision response. After the position of the CPlayer has been modified by the CPlayer::Update function, it loops through the m_pUpdatePlayer array (which contains one or more function call-backs) and calls each function in this array passing in a pointer to the CPlayer object itself. In our application, we add a single function call-back to the m_pUpdateArray: a pointer to the CTerrain::UpdatePlayer static function. When this function is called and passed the address of the CPlayer object, the terrain can check whether the CPlayer is intersecting it and then adjust the position of the player object so that it is positioned properly on top of the terrain. The same is true for the m_pUpdateCamera array. When the CPlayer::Update function moves the camera, it goes through the same procedure. It loops through the m_pUpdateCamera array and if any elements exist, the call-back function is called, this time passing in a pointer to the Camera so that the call-back function can modify its position.

We add function call-backs to the CALLBACK_FUNC arrays by calling the CPlayer::AddPlayerCallback and CPlayer::AddCameraCallback to add call-back functions for the CPlayer and the CCamera respectively. These functions are shown again below.

```
void        AddPlayerCallback       ( UPDATEPLAYER pFunc,  LPVOID pContext );
void        AddCameraCallback       ( UPDATECAMERA pFunc,  LPVOID pContext );
```

Our application calls these functions one time each during initialization. This provides our CPlayer object with one call-back for the CPlayer and one call-back for the CCamera. We pass in a pointer to the call-back function and a pointer to a user defined context. Let us take a look at the way these functions are called from the CGameApp::SetupGameState function to hopefully help keep things clear:

```
m_Player.AddPlayerCallback( CTerrain::UpdatePlayer,  (LPVOID)&m_Terrain );
m_Player.AddCameraCallback( CTerrain::UpdateCamera,  (LPVOID)&m_Terrain );
```

As you will see later, the CTerrain class has two functions called CTerrain::UpdatePlayer and CTerrain::UpdateCamera which are static methods of the CTerrain class. These functions are the call-back functions passed in to the above two functions that are added to the two CPlayer call-back arrays. Note that the function takes a void pointer to a context in parameter two. Our application passes the address of a CTerrain instance. These two pointers are stored in a CALLBACK_FUNC structure and added to the two call-back arrays.

In Chapter One we discussed the fact that call-back functions either have to be global functions or static class methods. Static class methods are like global functions, but are accessed as part of the class namespace. This means that the static function can always be called and always exists in memory even when instances of the class have not been created. A static class function can only access static member variables from the same class and is shared among all instances of that class. With this in mind, we need a way for the CTerrain static function to actually work with an instance of the terrain. This is why we pass in the address of an instance of the terrain class. That way, the pointer to the actual instance of the CTerrain class can be sent to the static CTerrain call-back functions during the CPlayer update. Although our application only uses a single instance of the CTerrain class, the static member function will need a pointer to an instance in order to access non-static class members. Another nice thing about making a call-back a static member of a class is that the call-back function can automatically access the private member variables of the instance passed in. The description of the call-back arrays will be covered in more detail later. Just bear in mind that these two arrays hold functions pointers which are called when the CPlayer moves so that classes external to the CPlayer/CCamera classes can commit to the position changes.

USHORT m_nUpdatePlayerCount
USHORT m_nUpdateCameraCount
These two variables hold the number of call-back functions that have been added to the **m_pUpdatePlayer** and the **m_pUpdateCamera** arrays respectively.

Let us now look at the member functions. As mentioned, many of the member functions are just variable assigner/retrievers which are inlined and shown in the above code. These functions will not be discussed.

CPlayer::CPlayer()

The first function we will look at is the CPlayer class constructor which assigns default values to the member variables. This is shown below.

```
CPlayer::CPlayer()
{
    // Clear any required variables
    m_pCamera            = NULL;
    m_pthirdPersonObject = NULL;
    m_CameraMode         = 0;

    // Initially no call-backs added to either array
    m_nUpdatePlayerCount = 0;
    m_nUpdateCameraCount = 0;

    // Players position & orientation (independant of camera)
    m_vecPos             = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    m_vecRight           = D3DXVECTOR3( 1.0f, 0.0f, 0.0f );
    m_vecUp              = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );
    m_vecLook            = D3DXVECTOR3( 0.0f, 0.0f, 1.0f );

    // Camera offset values (from the players origin)
    m_vecCamOffset       = D3DXVECTOR3( 0.0f, 10.0f, 0.0f );
    m_fCameraLag         = 0.0f;

    // The following force related values are used in conjunction with 'Update' only
    m_vecVelocity        = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    m_vecGravity         = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    m_fMaxVelocityXZ     = 125.0f;
    m_fMaxVelocityY      = 125.0f;
    m_fFriction          = 250.0f;

    // Set default bounding volume so it has no volume
    m_Volume.Min         =  D3DXVECTOR3 (0.0f , 0.0f , 0.0f);
    m_Volume.Max         =  D3DXVECTOR3 (0.0f , 0.0f , 0.0f);
 }
```

These initial values are mainly insignificant since we will want to set them up by calling the CPlayer member functions before using the camera.

CPlayer::~CPlayer()

The destructor deletes the attached camera if one exists. The CPlayer is responsible for creating the camera when we call CPlayer::SetCameraMode. Notice that it does not delete the CObject since this

class is not responsible for creating it. The CObject is created by the application and attached to the CPlayer class with a call to CPlayer::SetthirdPersonObject.

```
CPlayer::~CPlayer()
{
    // Release any allocated memory
    if ( m_pCamera ) delete m_pCamera;

    // Clear required values
    m_pCamera            = NULL;
    m_pthirdPersonObject = NULL;
}
```

CPlayer::SetCameraMode

CPlayer::SetCameraMode is the first CPlayer member function called by CGameApp::SetupGameState. This is called to initially place the CPlayer into first person camera mode. It is also called again in response to the user requesting a change of camera mode from the application menu. This function is responsible for releasing any previous cameras and creating a new camera object of the correct type.

This function first checks that the user is not selecting a camera mode that the CPlayer class is currently using. If so, it simply returns true. If this is not the case then we allocate a new camera object based on the type requested.

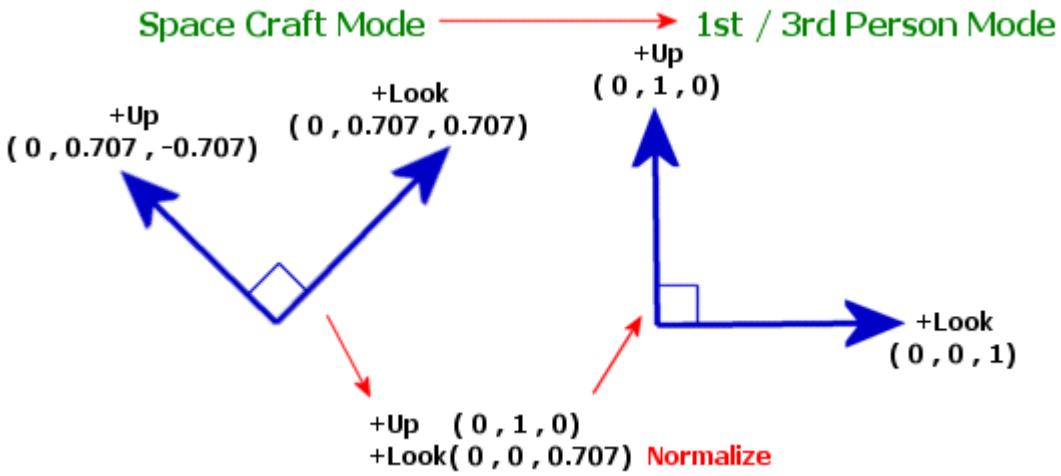
```
bool CPlayer::SetCameraMode( ULONG Mode )
{
    CCamera * pNewCamera = NULL;

    // Check for a no-op
    if ( m_pCamera && m_CameraMode == Mode ) return true;

    // Which mode are we switching into
    switch ( Mode )
    {
        case CCamera::MODE_FPS:
            if ( !(pNewCamera = new CCamfirstPerson( m_pCamera ))) return false;
            break;

        case CCamera::MODE_THIRDPERSON:
            if ( !(pNewCamera = new CCamthirdPerson( m_pCamera ))) return false;
            break;

        case CCamera::MODE_SPACECRAFT:
            if ( !(pNewCamera = new CCamSpaceCraft( m_pCamera ))) return false;
            break;
   }

    // Validate
    if (!pNewCamera) return false;
```

If the allocation fails, then we return false. Notice that because at this time we have not deleted the previous camera (if one exists) and have not changed the CCamera member pointer, we can return from the function failure, but still leave the current camera intact.

If the previous camera mode was spacecraft mode, we need to reset its local axis so that it is aligned to the XZ plane. Remember, in spacecraft mode we may have rotated the player completely upside down, so what we do before setting it into first or third person camera modes is zero out the pitch and roll values and reset the Y component of the Look and Right vectors such that the CPlayer is not pitched up -- which is not allowable in first or third person camera mode. By removing the Y component from the Look and Right vectors, we have made them non-unit length, so we will normalize them to make sure that they are. At this point we now have a set of up and look vectors that are parallel to the XZ plane and an Up vector that points directly up, aligned with the world Y axis.

```
    // If our old mode was SPACECRAFT we need to sort out some things
    if ( m_CameraMode == CCamera::MODE_SPACECRAFT )
    {
        // Flatten out the vectors
        m_vecUp      = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );
        m_vecRight.y = 0.0f;
        m_vecLook.y  = 0.0f;

        // Now normalize them
        D3DXVec3Normalize( &m_vecRight, &m_vecRight );
        D3DXVec3Normalize( &m_vecLook, &m_vecLook );
```

The following image is a side-on view of the CPlayer Up and Look vectors. The vectors in spaceship mode are rotated back and up 45 degrees respectively. Then we reset the Y vector to (0, 1, 0) and remove the Y component from the look vector. We normalize it to make it unit length and we now have a perpendicular set of axes which are no longer pitched.



In space craft mode the yaw, pitch, and roll values have no meaning since there is total freedom of rotation. But we must calculate what the Yaw angle is now that the axes have been reset if we are

going to first person mode. We use the dot product to measure the cosine of the angle between the new look vector and the world Z axis. We feed this into the acos function to convert this into an angle in radians. This is all inside the braces of the D3DXToDegree function so we will eventually get the yaw angle in degrees. The dot product returns the cosine of the angle between two vectors but does not tell us the relationship. So we check the x component of the look vector: if it is negative then we have a negative yaw angle. This means that we can tell the difference if the bearing from the look vector to the world z axis is 10 degrees or –10 degrees:

```
        // Reset our pitch / yaw / roll values
         m_fPitch = 0.0f;
         m_fRoll  = 0.0f;
         m_fYaw   = D3DXToDegree(acosf(D3DXVec3Dot(&D3DXVECTOR3(0.0f,0.0f,1.0f), \
                                                                  &m_vecLook)));
         if ( m_vecLook.x < 0.0f ) m_fYaw = -m_fYaw;
    }
```

The following image shows two look vectors (top-down view) that are oriented differently but which the dot product would return the same 45 degree angle for. We check the sign of the X component to determine which side of the world Z axis the new look vector points, giving us our positive or negative yaw angle.

If we are changing from another mode to space craft mode, then we need to synchronize the camera Up, Look, and Right vectors with the CPlayers Up, Look and Right vectors because in space craft mode, the camera is perfectly synchronized to the axes and rotations of the CPlayer class.

```
    else if (m_pCamera && Mode == CCamera::MODE_SPACECRAFT)
    {
        m_vecRight = m_pCamera->GetUp();
        m_vecLook  = m_pCamera->GetLook();
        m_vecUp    = m_pCamera->GetUp();
    }
```

Next we tell the newly created camera which CPlayer object it is attached to. We do this by calling the CCamera::AttachToPlayer function and pass in a pointer to this CPlayer object. We also store the new camera mode.

```
    // Store new mode
    m_CameraMode = Mode;

    // Attach the new camera to 'this' player object
    pNewCamera->AttachToPlayer( this );
```

Finally, we delete the old camera if one exists, and assign the member variable camera pointer to the newly created camera object.

```
    // Destroy our old camera and replace with our new one
    if ( m_pCamera ) delete m_pCamera;
    m_pCamera = pNewCamera;

    // Success!!
    return true;
}
```

Here is the function in its entirety:

```
bool CPlayer::SetCameraMode( ULONG Mode )
{
    CCamera * pNewCamera = NULL;
    if ( m_pCamera && m_CameraMode == Mode ) return true;

    // Which mode are we switching into
    switch ( Mode )
    {
        case CCamera::MODE_FPS:
            if ( !(pNewCamera = new CCamfirstPerson( m_pCamera ))) return false;
            break;

        case CCamera::MODE_THIRDPERSON:
           if ( !(pNewCamera = new CCamthirdPerson( m_pCamera ))) return false;
            break;

        case CCamera::MODE_SPACECRAFT:
           if ( !(pNewCamera = new CCamSpaceCraft( m_pCamera ))) return false;
            break;
    }
    if (!pNewCamera) return false;

    // If our old mode was SPACECRAFT we need to sort out some things
    if ( m_CameraMode == CCamera::MODE_SPACECRAFT )
    {
        // Flatten out the vectors
        m_vecUp      = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );
        m_vecRight.y = 0.0f;
        m_vecLook.y  = 0.0f;

        // Finally, normalize them
        D3DXVec3Normalize( &m_vecRight, &m_vecRight );
        D3DXVec3Normalize( &m_vecLook, &m_vecLook );

        // Reset our pitch / yaw / roll values
        m_fPitch = 0.0f;
        m_fRoll  = 0.0f;
        m_fYaw   = D3DXToDegree(acosf( D3DXVec3Dot(&D3DXVECTOR3(0.0f,0.0f,1.0f),
```

```
                                                                    &m_vecLook)));
        if ( m_vecLook.x < 0.0f ) m_fYaw = -m_fYaw;
    }
    else if (m_pCamera && Mode == CCamera::MODE_SPACECRAFT)
    {
         m_vecRight = m_pCamera->GetUp();
         m_vecLook  = m_pCamera->GetLook();
         m_vecUp    = m_pCamera->GetUp();
    }

    // Store new mode
    m_CameraMode = Mode;

    // Attach the new camera to 'this' player object
    pNewCamera->AttachToPlayer( this );

    // Destroy our old camera and replace with our new one
    if ( m_pCamera ) delete m_pCamera;
    m_pCamera = pNewCamera;

    // Success!!
    return true;
}
```

CPlayer::AddPlayerCallback
CPlayer::AddCameraCallback

These functions add call-back functions to the two call-back arrays. We will show the code only to the AddPlayerCallback function here as the AddCameraCallback code is exactly the same, with the exception that it adds the CALLBACK_FUNC structure to the **m_pCameraUpdate** array instead of the **m_pPlayerUpdate** array.

```
void CPlayer::AddPlayerCallback( UPDATEPLAYER pFunc, LPVOID pContext )
{
    // Store callback details
    m_pUpdatePlayer[m_nUpdatePlayerCount].pFunction = (LPVOID)pFunc;
    m_pUpdatePlayer[m_nUpdatePlayerCount].pContext  = pContext;
    m_nUpdatePlayerCount++;
}
```

This function takes as its first parameter a pointer to a call-back function and as its second parameter a void pointer to the associated context that you would like to have passed to the call-back function when it is called. In our application, the first parameter is a pointer to the CTerrain::UpdatePlayer function and the second parameter is a pointer to the actual instance of the terrain that is maintained by the CGameApp class.

Function Pointers

For those of you not familiar with function pointers, the UPDATEPLAYER type is typedef'd in the CPlayer.h file as:

typedef void (*UPDATEPLAYER)(LPVOID pContext, CPlayer *pPlayer, float TimeScale);

This means that we can declare variables to be of type UPDATEPLAYER as shown below.

```
UPDATEPLAYER MyFuncPointer;
```

MyFuncPointer is now a pointer to a function that returns void and accepts the three parameters shown above. Once we declare a function pointer, we can assign it the address of a pointer stored in our call-back array and ultimately call that function using the pointer.

The following code shows how this type of pointer is used in the CPlayer::Update function to call a function that is stored in the m_pPlayerUpdate array:

```
UPDATEPLAYER UpdatePlayer = (UPDATEPLAYER)m_pUpdatePlayer[i].pFunction;
```

And finally, we can call that function:

```
UpdatePlayer( m_pUpdatePlayer[i].pContext, this, TimeScale );
```

CPlayer::SetCamOffset

The next function of consequence called from CGameApp::SetupGameState is CPlayer::SetCamOffset. It allows us to specify where the camera is to be placed in relation to the CPlayer object. In first person mode our application sets this vector to (0, 10, 0) so that the camera is placed 10 units above the CPlayer object. This function is also called whenever the user changes camera modes from the application window menu. If changing to space craft mode, the offset is set to zero. When changing to third person mode, we set the offset vector to (0, 40, -60) so the camera is always tracking the CPlayer object from a distance of 40 units above and 60 units behind.

The function sets the internal member variable to the passed offset and then calls the camera class function SetPosition to set the position of the camera to the new position. Remember that the position of the camera should be the position of the CPlayer plus the offset vector. This means if the CPlayer was currently at world space position (0, 500, 1000) and we passed an offset vector of (0, 50, -100) then the camera would be positioned at (0 , 550 , 900). We will cover the camera class functions later.

```
void CPlayer::SetCamOffset( const D3DXVECTOR3& Offset )
{
    m_vecCamOffset = Offset;

    if (!m_pCamera) return;
```

```
    m_pCamera->SetPosition( m_vecPos + Offset );
}
```

Those are all of the set-up functions that we need to cover for the CPlayer object. Let us now move on to the functions that are called from within the main game loop to update the CPlayer position and orientation.

CGameApp::ProcessInput is called every frame to get the state of the keyboard and mouse and to determine whether any rotations need to occur. If the mouse is moved left or right, or up and down, then the mouse movement is turned into degrees and CPlayer::Rotate is called with the desired rotation angles.

CPlayer::Rotate

This function works differently depending on the camera mode so the first thing we do is get the attached camera's current mode to check it.

```
void CPlayer::Rotate( float x, float y, float z )
{
    D3DXMATRIX mtxRotate;

    // Validate requirements
    if (!m_pCamera) return;

    // Retrieve camera mode
    CCamera::CAMERA_MODE Mode = m_pCamera->GetCameraMode();
```

If we are in first person mode or third person mode, then the rotations are applied differently than if we are in space craft mode. The next section shows the code executed when we are not in space craft mode.

```
    if ( Mode == CCamera::MODE_FPS || Mode == CCamera::MODE_THIRDPERSON )
    {
        // update & clamp pitch / roll / yaw values
        if ( x )
        {
            // Make sure we don't overstep our pitch boundaries
            m_fPitch += x;
            if ( m_fPitch >  89.0f ) { x -= (m_fPitch - 89.0f); m_fPitch = 89.0f; }
            if ( m_fPitch < -89.0f ) { x -= (m_fPitch + 89.0f); m_fPitch = -89.0f; }
        }
```

The first thing we do is add the rotation angle to our current pitch value. The pitch value is only used in first person mode and is used to rotate the camera up and down. The pitch range is 89 degrees in both directions so we must clamp the pitch to 89 or –89 depending on whether we are rotating up or down. All we have done at this point is add the x rotation angle to our pitch value. The pitch value is never used to pitch the CPlayer object but will be forwarded on to the attached camera object. If the camera

is a first person camera, it will use this value to rotate itself about its own Right vector. This value is ignored if the attached camera is a third person camera.

Next we do the same with the Y angle by adding it to the current yaw value, but we do not clamp the result in this case. In both first and third person camera modes, the CPlayer object is allowed to rotate endlessly about its Up vector. We do make sure to roll the value back around again if it exceeds the 0 – 360 degree range. If we rotate past 360 it becomes zero again and vice versa

```
        if ( y )
        {
            // Ensure yaw (in degrees) wraps around between 0 and 360
            m_fYaw += y;
            if ( m_fYaw >  360.0f ) m_fYaw -= 360.0f;
            if ( m_fYaw <  0.0f   ) m_fYaw += 360.0f;
        }
```

Finally we do the same for the Z axis rotation, but this time we clamp the value to 20 degrees in each direction. This value is not used by the CPlayer class but is forwarded to the attached camera class. If the camera is a third person camera, this value is ignored. If the camera is a first person camera, it uses this angle to rotate the camera about the player's Look vector to perform a lean.

```
        if ( z )
        {
            // Make sure we don't overstep our roll boundaries
            m_fRoll += z;
            if ( m_fRoll >  20.0f ) { z -= (m_fRoll - 20.0f); m_fRoll = 20.0f; }
            if ( m_fRoll < -20.0f ) { z -= (m_fRoll + 20.0f); m_fRoll = -20.0f; }
        }
```

Now that we have added our rotation values to the internal values and clamped them to their limits, we pass these angles to the attached camera class by calling the CCamera::Rotate function:

```
        // Allow camera to rotate prior to updating our axis
        m_pCamera->Rotate( x, y, z );
```

If the attached camera is a first person camera, the camera will rotate about the proper axis by the specified amount. If we are using a third person camera, the Rotate function is empty and does nothing. This is because we never actually rotate the third person camera; it is automatically adjusted to always look at the CPlayer object.

Finally, we use the Y value to rotate the CPlayer object itself by rotating its Look and Right vectors about its Up vector. This is because in both first and third person modes, the Y rotation will yaw the player object around his Up vector.

```
        // Now rotate our axis
        if ( y )
        {
            // Build rotation matrix
            D3DXMatrixRotationAxis( &mtxRotate, &m_vecUp, D3DXToRadian( y ) );
```

```
            // Update our vectors
            D3DXVec3TransformNormal( &m_vecLook, &m_vecLook, &mtxRotate );
            D3DXVec3TransformNormal( &m_vecRight, &m_vecRight, &mtxRotate );
        }
    } // End if MODE_firstPerson or MODE_thirdPerson
```

If we are in spacecraft mode then the rotation code is different because the player can be rotated about all three axes. In this mode, the camera and the player are rotated in sync. This is the same code as the example rotation function we looked at in the textbook which showed how us to rotate the camera about its own axes.

```
    else if ( Mode == CCamera::MODE_SPACECRAFT )
    {
        // Allow camera to rotate prior to updating our axis
        m_pCamera->Rotate( x, y, z );

        if ( x != 0 )
        {
            // Build rotation matrix
            D3DXMatrixRotationAxis( &mtxRotate, &m_vecRight, D3DXToRadian( x ) );
            D3DXVec3TransformNormal( &m_vecLook, &m_vecLook, &mtxRotate );
            D3DXVec3TransformNormal( &m_vecUp, &m_vecUp, &mtxRotate );
        }

        if ( y != 0 )
        {
            // Build rotation matrix
            D3DXMatrixRotationAxis( &mtxRotate, &m_vecUp, D3DXToRadian( y ) );
            D3DXVec3TransformNormal( &m_vecLook, &m_vecLook, &mtxRotate );
            D3DXVec3TransformNormal( &m_vecRight, &m_vecRight, &mtxRotate );
        }

        if ( z != 0 )
        {
            // Build rotation matrix
            D3DXMatrixRotationAxis( &mtxRotate, &m_vecLook, D3DXToRadian( z ) );
            D3DXVec3TransformNormal( &m_vecUp, &m_vecUp, &mtxRotate );
            D3DXVec3TransformNormal( &m_vecRight, &m_vecRight, &mtxRotate );

        }
    }// end space craft
```

At this point, the camera and the CPlayer object have had their Up, Look, and Right vectors rotated to represent their new orientations. All we have to do now before we return is perform vector regeneration on the CPlayer axes to prevent floating point accumulation errors from creeping in.

```
    // Vector regeneration
    D3DXVec3Normalize( &m_vecLook, &m_vecLook );
    D3DXVec3Cross( &m_vecRight, &m_vecUp, &m_vecLook );
    D3DXVec3Normalize( &m_vecRight, &m_vecRight );
    D3DXVec3Cross( &m_vecUp, &m_vecLook, &m_vecRight );
    D3DXVec3Normalize( &m_vecUp, &m_vecUp );
}
```

CPlayer::Move

Recall that in the ProcessInput function, after and rotations have been made, the current state of the keys are recorded in a DWORD bit set and sent to CPlayer::Move:

```
m_Player.Move( Direction, 500.0f * m_Timer.GetTimeElapsed(), true );
```

The first parameter holds the bit set and the second parameter is the acceleration we wish to apply. We also pass in a Boolean to determine whether we wish to work through the velocity vector of the CPlayer (true) or rather to instantly displace the player by this amount (false). Our application passes true, which means that it is added to the velocity vector and will have gravity and friction applied to create a final velocity vector.

This function does not actually apply any movement to the camera in 'true' mode, it simply calculates a direction vector based on keys pressed and acceleration applied.

```
void CPlayer::Move( ULONG Direction, float Distance, bool Velocity )
{
    D3DXVECTOR3 vecShift = D3DXVECTOR3( 0, 0, 0 );

    // Which direction are we moving ?
    if ( Direction & DIR_FORWARD  ) vecShift += m_vecLook  * Distance;
    if ( Direction & DIR_BACKWARD ) vecShift -= m_vecLook  * Distance;
    if ( Direction & DIR_RIGHT    ) vecShift += m_vecRight * Distance;
    if ( Direction & DIR_LEFT     ) vecShift -= m_vecRight * Distance;
    if ( Direction & DIR_UP       ) vecShift += m_vecUp    * Distance;
    if ( Direction & DIR_DOWN     ) vecShift -= m_vecUp    * Distance;

    // Update camera vectors
    if ( Direction ) Move( vecShift, Velocity );
}
```

Assume we have a player facing down the world X axis -- his Look vector would be (1,0,0) and his Up vector would be (0,1,0). If we press both the left and up keys and pass in an acceleration of 350, the direction vector created would be (-1 * 350 , 1 * 350 , 0) = (-350 , 350 ,0). We now pass this vector to an overloaded version of the Move function shown next. If we have passed true to the previous function, then the direction vector is added to the CPlayer current velocity vector. If we pass false, then the CPlayer and the attached camera are instantly moved along this vector into their new positions.

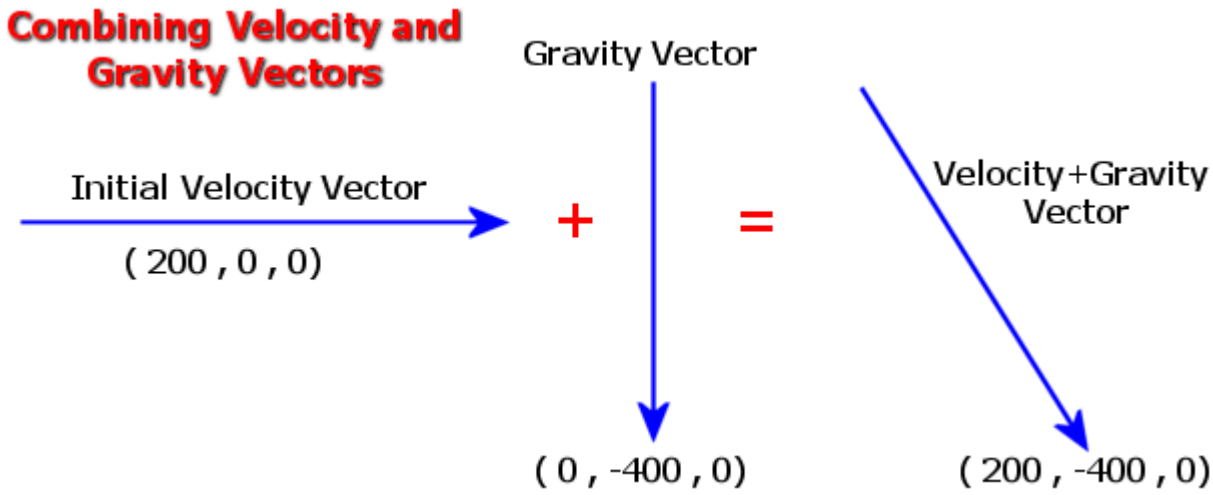
```
void CPlayer::Move( const D3DXVECTOR3& vecShift, bool Velocity )
{
    // Update velocity or actual position ?
    if ( Velocity ){
        m_vecVelocity += vecShift;
    }
    else{
        m_vecPos += vecShift;
        m_pCamera->Move( vecShift );
    }
 }
```

CPlayer::Update

This function will be called every frame to apply friction and gravity to the velocity vector, to move the CPlayer and its attached camera to its new position along the velocity vector, and to give the camera a final chance to update itself before the frame is drawn.

```
void CPlayer::Update( float TimeScale )
{
    // Add on our gravity vector
    m_vecVelocity += m_vecGravity * TimeScale;
```

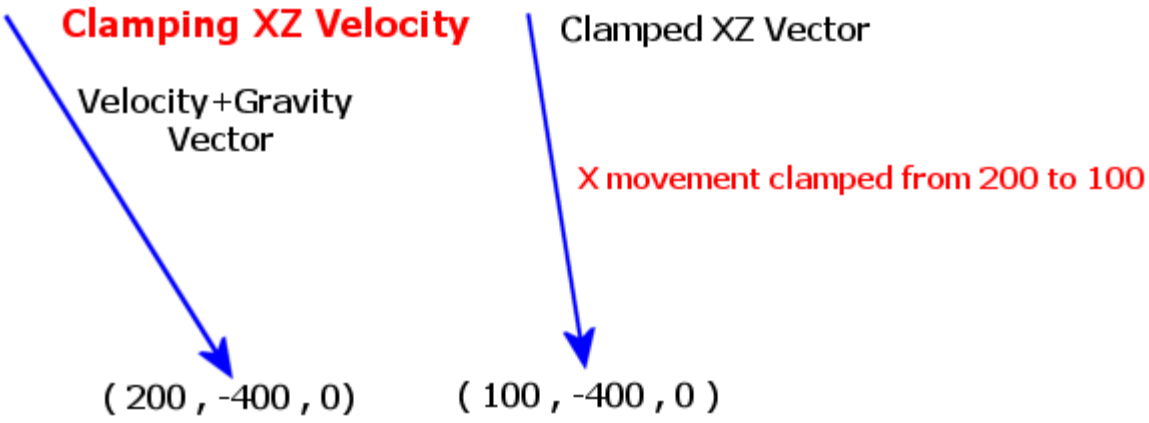
The first thing we do is apply gravity to the velocity vector. In our demo, gravity is a vector pointing vertically down with a magnitude of 400 world units.



Before we apply the new velocity to the CPlayer position, we make sure that we are not moving further in the XZ plane than is permitted in the elapsed time. Therefore, we calculate the length of just the X and Z components of the velocity vector and clamp them to their maximum ranges if necessary.

```
    // Clamp the XZ velocity to our max velocity vector
    float Length = sqrtf(m_vecVelocity.x * m_vecVelocity.x +
                         m_vecVelocity.z * m_vecVelocity.z);
    if ( Length > m_fMaxVelocityXZ )
    {
        m_vecVelocity.x *= ( m_fMaxVelocityXZ / Length );
        m_vecVelocity.z *= ( m_fMaxVelocityXZ / Length );
    }
```

We only clamp the XZ components because we are allowed to have a different maximum velocity in the Y dimension. Using the vectors in the above image as an example, if we had a maximum XZ velocity of 100, then the velocity vector would be clamped because it is currently moving 200 units in the X dimension.



Next we clamp the Y component of the velocity vector if it exceeds the maximum allowed range:

```
    // Clamp the Y velocity to our max velocity vector
    Length = sqrtf(m_vecVelocity.y * m_vecVelocity.y);
    if ( Length > m_fMaxVelocityY )
    {
        m_vecVelocity.y *= ( m_fMaxVelocityY / Length );
    }
```

We now have a vector that describes the direction and distance that we would like to move. We call CPlayer::Move and pass in the velocity vector and a flag value of false so that the velocity vector is directly added to the position of the CPlayer and Camera objects.

```
    // Move our player (will also move the camera if required)
    Move( m_vecVelocity * TimeScale, false );
```

One interesting thing about the above function call is that it moves the player and then moves the attached camera by calling the camera's Move function. This function body is implemented in both the first person and space craft camera classes which causes the camera to move in sync with the player. If the attached camera is a third person camera however, the Move function is empty and does nothing,

At this point, the player has moved itself into his new position, but the CPlayer object has no concept of the CTerrain class or that the terrain even exists. This means of course, that the player may have moved themselves right into a mountain or some other such illegal place. In our example, the CTerrain class has added a call-back function to the internal call-back arrays, so the next job of the Update function is to loop through each element in the m_pPlayerUpdate array and call each call-back function that has been added to this array. There is only one function in this array in our demo and that is the CTerrain::UpdatePlayer function. This function is passed the CPlayer and can check its position against the terrain. The nice thing about this system is that virtually any routine can use the CPlayer object without the CPlayer object having any knowledge of the scene geometry. As long as the scene geometry database provides a call-back function (or perhaps a few call-backs) the CPlayer will call it in its update function allowing for position modification. Once again we strongly emphasize that this is not a robust or recommended collision determination system – it simply serves our purposes for these

small demonstrations. You will more likely use a higher level collision manager as part of a larger Physics engine that handles all object-object and object-environment interaction. The Game Institute offers training in Physics for game development so be sure to check out the course when you are ready to increase the capabilities of your engine.

```
    // Allow all our registered call-backs to update the player position
    for ( i =0; i < m_nUpdatePlayerCount; i++ )
    {
        UPDATEPLAYER UpdatePlayer = (UPDATEPLAYER)m_pUpdatePlayer[i].pFunction;
        UpdatePlayer( m_pUpdatePlayer[i].pContext, this, TimeScale );
    }
```

At this point, the player has been moved to their new position and so has the camera -- provided it is not a third person camera; in which case it is still unaltered. Next we call the CCamera::Update function to give the attached camera class an opportunity to modify itself in its new position. This function does nothing in the first person and spacecraft camera modes but it does have an implementation in third person mode. It is in the CCamthirdPerson::Update function that the new position of the CPlayer is retrieved to calculate where the third person camera should move. Each time the update function is called, the third person camera moves slowly (depending on camera lag) into its desired position. It also ensures the camera is looking at the CPlayer.

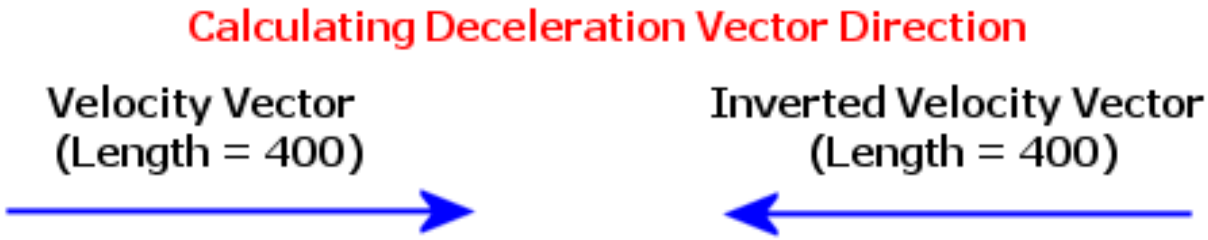
```
    // Let our camera update if required
    m_pCamera->Update( TimeScale, m_fCameraLag );
```

As with the CPlayer object, the camera may have been moved into an illegal position with respect to the scene geometry. As we did with the **m_pUpdatePlayer** array, we now loop through the **m_pUpdateCamera** and call every call-back function contained within. This gives external classes a chance to modify/correct the camera position.

```
    for ( i =0; i < m_nUpdateCameraCount; i++ )
    {
        UPDATECAMERA UpdateCamera = (UPDATECAMERA)m_pUpdateCamera[i].pFunction;
        UpdateCamera( m_pUpdateCamera[i].pContext, m_pCamera, TimeScale );
    }
```

Before we leave this function, we will apply the friction/drag coefficient to the velocity vector for deceleration. If we did not do this, the player would carry on moving forever along the velocity vector. In order to do this, we create a deceleration vector and add it to the velocity vector at the end of the update. To calculate the deceleration vector, we create a vector that points in the opposite direction of the velocity vector and store the result in another vector called vecDec.

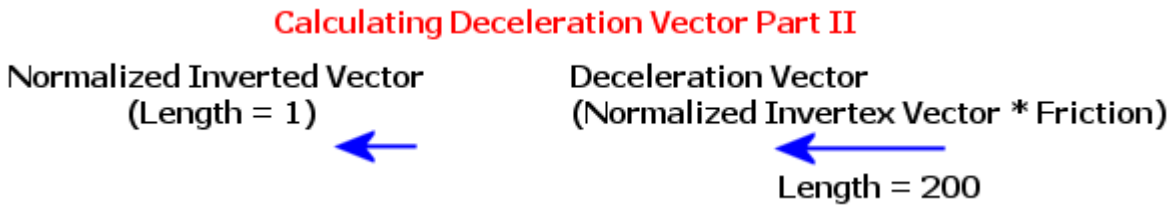
```
    // Calculate the reverse of the velocity direction
    D3DXVECTOR3 vecDec = -m_vecVelocity;
```



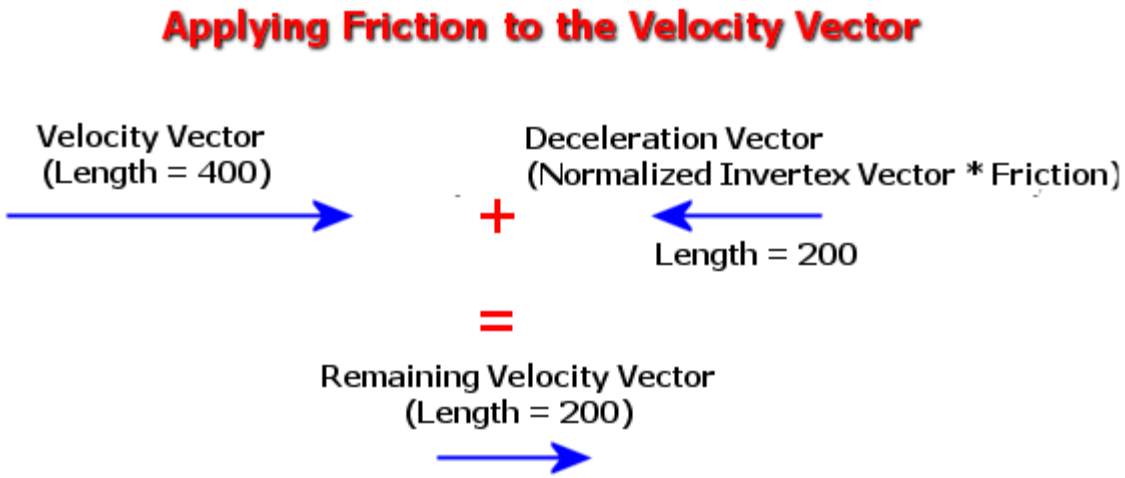
We now scale the inverted velocity vector such that it has a length that is equal to the friction/drag value we have set. We do this by normalizing the inverted vector so that it has a length of 1, and then multiply it by the friction/drag value so that its length is equal to that value.

```
    // Normalize the deceleration vector
    D3DXVec3Normalize( &vecDec, &vecDec );

    // Retrieve the actual velocity length
    Length = D3DXVec3Length( &m_vecVelocity );

    // Calculate total deceleration based on friction values
    float Dec = (m_fFriction * TimeScale);
    if ( Dec > Length ) Dec = Length;

    // Apply the friction force
    m_vecVelocity += vecDec * Dec;
}
```

The next time this function is called, the velocity vector will have decreased (assuming the application has not requested additional acceleration). Remember that acceleration is always applied when the user is holding down one of the movement keys. The following image depicts scaling the unit deceleration vector by a drag coefficient of 200 units.



Finally, this deceleration is added to the velocity vector so that its length is diminished.



Here is the Update function in its entirety:

```
void CPlayer::Update( float TimeScale )
{
    // Add on our gravity vector
    m_vecVelocity += m_vecGravity * TimeScale;

    // Clamp the XZ velocity to our max velocity vector
    float Length = sqrtf(m_vecVelocity.x * m_vecVelocity.x +
                         m_vecVelocity.z * m_vecVelocity.z);
    if ( Length > m_fMaxVelocityXZ ){
        m_vecVelocity.x *= ( m_fMaxVelocityXZ / Length );
        m_vecVelocity.z *= ( m_fMaxVelocityXZ / Length );
    }

    // Clamp the Y velocity to our max velocity vector
    Length = sqrtf(m_vecVelocity.y * m_vecVelocity.y);
    if ( Length > m_fMaxVelocityY ){
        m_vecVelocity.y *= ( m_fMaxVelocityY / Length );
    }

    // Move our player (will also move the camera if required)
    Move( m_vecVelocity * TimeScale, false );

    // Allow all our registered callbacks to update the player position
    for ( i =0; i < m_nUpdatePlayerCount; i++ )
    {
        UPDATEPLAYER UpdatePlayer = (UPDATEPLAYER)m_pUpdatePlayer[i].pFunction;
        UpdatePlayer( m_pUpdatePlayer[i].pContext, this, TimeScale );
    }

    // Let our camera update if required
    m_pCamera->Update( TimeScale, m_fCameraLag );

    // Allow all our registered callbacks to update the camera position
    for ( i =0; i < m_nUpdateCameraCount; i++ )
    {
        UPDATECAMERA UpdateCamera = (UPDATECAMERA)m_pUpdateCamera[i].pFunction;
```

```
        UpdateCamera( m_pUpdateCamera[i].pContext, m_pCamera, TimeScale );
    }

    // Calculate the reverse of the velocity direction
    D3DXVECTOR3 vecDec = -m_vecVelocity;
    D3DXVec3Normalize( &vecDec, &vecDec );

    // Retrieve the actual velocity length
    Length = D3DXVec3Length( &m_vecVelocity );

    // Calculate total deceleration based on friction values
    float Dec = (m_fFriction * TimeScale);
    if ( Dec > Length ) Dec = Length;

    // Apply the friction force
    m_vecVelocity += vecDec * Dec;
}
```

CPlayer::Render

This function is responsible for rendering the attached CObject when we are in third person mode. In our application this is a simple cube which is rendered as an indexed triangle strip using a single call to DrawIndexedPrimitive. In order to render the CObject in its correct position, we must create a world matrix. As you will see, we build this matrix using the CPlayer Right, Up, Look and position vectors. We then set the matrix as the device world matrix and render the CObject mesh.

```
void CPlayer::Render( LPDIRECT3DDEVICE9 pDevice )
{
    CObject * pObject = NULL;

    // Select which object to render
    if ( m_pCamera )
    {
        if ( m_CameraMode == CCamera::MODE_THIRDPERSON ) pObject = m_pthirdPersonObject;
    }
    else
    {
        // Select the third person object (viewed from outside)
        pObject = m_pthirdPersonObject;
    }

    // Validate
    if (!pObject) return;

    // Update our object's world matrix
    D3DXMATRIX * pMatrix = &pObject->m_mtxWorld;
    pMatrix->_11 = m_vecRight.x; pMatrix->_21 = m_vecUp.x; pMatrix->_31 = m_vecLook.x;
    pMatrix->_12 = m_vecRight.y; pMatrix->_22 = m_vecUp.y; pMatrix->_32 = m_vecLook.y;
    pMatrix->_13 = m_vecRight.z; pMatrix->_23 = m_vecUp.z; pMatrix->_33 = m_vecLook.z;
    pMatrix->_41 = m_vecPos.x;
    pMatrix->_42 = m_vecPos.y - 10.0f;
    pMatrix->_43 = m_vecPos.z;

    // Render our player mesh object
```

```
    CMesh * pMesh = pObject->m_pMesh;
    pDevice->SetTransform( D3DTS_WORLD, &pObject->m_mtxWorld );
    pDevice->SetStreamSource( 0, pMesh->m_pVertexBuffer, 0, sizeof(CVertex) );
    pDevice->SetIndices( pMesh->m_pIndexBuffer );
    pDevice->DrawIndexedPrimitive( D3DPT_TRIANGLESTRIP, 0, 0, 8, 0, 14 );
}
```

The local scope CObject pointer is not assigned to the third person object if we are using any camera other than third person. If a camera is attached to the CPlayer but it is not a third person camera, then the CObject pointer remains a NULL pointer and the function returns without rendering the model.

Next we will examine the various camera classes at our disposal. The CCamera class is very straightforward, and the three derived classes (CCamfirstPerson, CCamthirdPerson, and CCamSpaceCraft) simply override a handful of virtual functions to provide different behaviours.

The CCamera Base Class

The CCamera class manages the view matrix as well as the projection matrix. For the view matrix, it will need to maintain a camera position and the Look, Up, and Right vectors. This means it will need to provide functions that allow the application to set the position and orientation of the camera. We have already seen these functions (CCamera::Move and CCamera::Rotate) called from the CPlayer class. To encapsulate the building and management of the projection matrix, we need variables that contain information such as the current field of view and the positions of the near and far clip planes. We will also need functions that allow us to attach or detach this camera to/from a CPlayer object. Finally, the camera class will maintain a bounding volume much like the CPlayer class. It is used by the CTerrain class to check whether the camera has collided with scene geometry.

Below we see the CCamera class declaration contained in the CCamera.h file. Many of these functions set/get member variables and as such their bodies are inlined in the header file. Many of the functions are also declared as virtual functions that will be overridden in derived classes. Some of these are simply empty functions in the base class (such as the Rotate, Move and Update functions). These are the functions that we will override to give specific functionality to the derived classes.

```
class CCamera
{
public:
    // Enumerator
    enum CAMERA_MODE {
        MODE_FPS         = 1,
        MODE_THIRDPERSON = 2,
        MODE_SPACECRAFT  = 3,
        MODE_FORCE_32BIT = 0x7FFFFFFF
    };

    // Constructors & Destructors for This Class.
    CCamera( const CCamera * pCamera );
    CCamera();
    virtual ~CCamera();
```

```
    // Public Functions for This Class.
    void  SetFOV        (float FOV) { m_fFOV = FOV; m_bProjDirty = true; }
    void  SetViewport   (long Left, long Top, long Width, long Height, float NearClip,
                         float FarClip, LPDIRECT3DDEVICE9 pDevice = NULL );

    void  UpdateRenderView  ( LPDIRECT3DDEVICE9 pD3DDevice );
    void  UpdateRenderProj  ( LPDIRECT3DDEVICE9 pD3DDevice );

    const D3DXMATRIX&   GetProjMatrix  ( );
    float               GetFOV         ( ) const    { return m_fFOV;  }
    float               GetNearClip    ( ) const    { return m_fNearClip; }
    float               GetFarClip     ( ) const    { return m_fFarClip; }
    const D3DVIEWPORT9& GetViewport    ( ) const { return m_Viewport; }
    CPlayer *           GetPlayer      ( ) const { return m_pPlayer;  }
    const D3DXVECTOR3&  GetPosition    ( ) const { return m_vecPos;   }
    const D3DXVECTOR3&  GetLook        ( ) const { return m_vecLook;  }
    const D3DXVECTOR3&  GetUp          ( ) const { return m_vecUp;    }
    const D3DXVECTOR3&  GetRight       ( ) const { return m_vecRight; }
    const D3DXMATRIX&   GetViewMatrix  ( );
    void                SetVolumeInfo  ( const VOLUME_INFO& Volume );
    const VOLUME_INFO&  GetVolumeInfo  ( ) const;

    // public virtual functions
    virtual void   AttachToPlayer   ( CPlayer * pPlayer );
    virtual void   DetachFromPlayer ( );
    virtual void   SetPosition(const D3DXVECTOR3& Position)
                                    {m_vecPos = Position; m_bViewDirty = true;}
    virtual void   Move(const D3DXVECTOR3& vecShift)
                                   { m_vecPos += vecShift; m_bViewDirty = true; }
    virtual void   Rotate( float x, float y, float z )   {}
    virtual void   Update( float TimeScale, float Lag )  {}
    virtual void   SetCameraDetails( const CCamera * pCamera )     {}
    virtual CAMERA_MODE GetCameraMode( ) const = 0;

    protected: // Member Variables
    CPlayer        * m_pPlayer;        // The player object we are attached to
    VOLUME_INFO      m_Volume;         // Stores information about cameras collision volume
    D3DXMATRIX       m_mtxView;        // Cached view matrix
    D3DXMATRIX       m_mtxProj;        // Cached projection matrix
    bool             m_bViewDirty;     // View matrix dirty ?
    bool             m_bProjDirty;     // Proj matrix dirty ?

    // Perspective Projection parameters
    float           m_fFOV;            // FOV Angle
    float           m_fNearClip;       // Near Clip Plane Distance
    float           m_fFarClip;        // Far Clip Plane Distance
    D3DVIEWPORT9    m_Viewport;        // The viewport details into which we are rendering

    // Cameras current position & orientation
    D3DXVECTOR3     m_vecPos;           // Camera Position
    D3DXVECTOR3     m_vecUp;            // Camera Up Vector
    D3DXVECTOR3     m_vecLook;          // Camera Look Vector
    D3DXVECTOR3     m_vecRight;         // Camera Right Vector
};
```

Most of the Set()/Get() functions are implemented in the header file and we will not cover these since their behaviour is obvious. First we will take a look at the member variables and their purpose.

CPlayer *m_Player
This is a pointer to a CPlayer object which the camera may be attached to. This is initialized to NULL.

VOLUME_INFO m_Volume
This is used to describe the bounding volume of the camera (an axis aligned bounding box in our application).

D3DXMATRIX m_mtxView
This is a 4x4 matrix used to hold the current state of the view matrix.

D3DXMATRIX m_mtxProj
This is a 4x4 matrix used to hold the current state of the projection matrix.

Bool m_bViewDirty, m_bProjDirty
These two Boolean variables are used to indicate that alterations have been made to the camera class that require the projection matrix or the view matrix to be rebuilt. For example, when we call the CCamera::SetPosition function to modify the position of the camera, the view matrix is not instantly rebuilt; instead the **m_bViewDirty** flag is set to true. This allows us to make several sequential changes to the camera without the cost of rebuilding the matrix each time. When we wish to set the view matrix as the device view matrix, we call the CCamera::GetViewMatrix function. If this flag is set to true, then it recalculates the new view matrix before returning it. If it has not been modified since the last call to CCamera::GetViewMatrix, then we can just return the currently cached copy. The m_bProjDirty function works the same way with regards to the projection matrix.

float m_fFOV
This value contains the angle of the current field of view (FOV) in degrees. It is used when the projection matrix needs to be rebuilt. We set the camera FOV by calling the CCamera::SetFOV function.

float m_fNearPlane and m_fFarPlane
These values contain the view space distance to the near and far planes of the projection matrix. We set these values in the call to CCamera::SetViewport.

D3DVIEWPORT9 m_Viewport
This is the desired rendering viewport for the camera.

D3DXVECTOR3 m_vecPos
This vector contains the current world space position of the camera.

D3DXVECTOR3 m_vecRight
D3DXVECTOR3 m_vecUp
D3DXVECTOR3 m_vecLook
These three vectors describe the orientation of the camera local Look, Up and Right vectors. They define the camera local coordinate system axes.

CCamera::CCamera()

There are two constructors for our camera. The first simply initializes all values to a good set of defaults. The view and projection matrices are set to identity and the Right, Up, and Look vectors are aligned with the world X, Y and Z axes respectively. The CPlayer pointer is set to null because the object is not yet attached to a CPlayer object. The field of view is initialized to 60 degrees (a nice default value) and the near and far clip planes are set at a distance of 1.0 and 100.0 from the camera respectively. The default viewport is 640x480 pixels and is positioned so that its top-left corner is located at coordinate (0,0) in the frame buffer.

```
CCamera::CCamera()
{
    // Reset / Clear all required values
    m_pPlayer         = NULL;
    m_vecRight        = D3DXVECTOR3( 1.0f, 0.0f, 0.0f );
    m_vecUp           = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );
    m_vecLook         = D3DXVECTOR3( 0.0f, 0.0f, 1.0f );
    m_vecPos          = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );

    m_fFOV            = 60.0f;
    m_fNearClip       = 1.0f;
    m_fFarClip        = 100.0f;
    m_Viewport.X      = 0;
    m_Viewport.Y      = 0;
    m_Viewport.Width  = 640;
    m_Viewport.Height = 480;
    m_Viewport.MinZ   = 0.0f;
    m_Viewport.MaxZ   = 1.0f;

    // Set matrices to identity
    D3DXMatrixIdentity( &m_mtxView );
    D3DXMatrixIdentity( &m_mtxProj );
}
```

The second constructor takes a pointer to a CCamera class to allow derived classes to initialize themselves based on the settings of previously created cameras. It is similar to a typical copy constructor.

```
CCamera::CCamera( const CCamera * pCamera )
{
    // Reset / Clear all required values
    m_pPlayer         = NULL;
    m_vecRight        = D3DXVECTOR3( 1.0f, 0.0f, 0.0f );
    m_vecUp           = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );
    m_vecLook         = D3DXVECTOR3( 0.0f, 0.0f, 1.0f );
    m_vecPos          = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );

    m_fFOV            = 60.0f;
    m_fNearClip       = 1.0f;
    m_fFarClip        = 100.0f;
    m_Viewport.X      = 0;
    m_Viewport.Y      = 0;
    m_Viewport.Width  = 640;
```

```
    m_Viewport.Height = 480;
    m_Viewport.MinZ   = 0.0f;
    m_Viewport.MaxZ   = 1.0f;

    // Set matrices to identity
    D3DXMatrixIdentity( &m_mtxView );
    D3DXMatrixIdentity( &m_mtxProj );
}
```

CCamera::~CCamera()

The CCamera class does not allocate any memory that needs to be released. Therefore, the default destructor has no function body and does nothing. Note that we use a virtual destructor so that if the object is destroyed via a pointer to the base class, the destructor for the object will be called correctly.

CCamera::SetViewport

CCamera::SetViewport takes input parameters for the viewport left (X) and top (Y) coordinates as well as its width and height. These values are copied into the D3DVIEWPORT9 member variable. MinZ and MaxZ are hard coded to 0.0 and 1.0 respectively and you will likely never need to change this. The last two parameters we pass are the distances to the near and far clip planes. These are not actually related to setting the viewport itself, but they do influence the projection matrix that will need to be recalculated.

```
void CCamera::SetViewport( long Left, long Top, long Width, long Height, float NearClip,
                           float FarClip, LPDIRECT3DDEVICE9 pDevice )
{
    // Set viewport sizes
    m_Viewport.X      = Left;
    m_Viewport.Y      = Top;
    m_Viewport.Width  = Width;
    m_Viewport.Height = Height;
    m_Viewport.MinZ   = 0.0f;
    m_Viewport.MaxZ   = 1.0f;
    m_fNearClip       = NearClip;
    m_fFarClip        = FarClip;
    m_bProjDirty      = true;

    // Update device if requested
    if ( pDevice ) pDevice->SetViewport( &m_Viewport );
}
```

The final parameter is a pointer to an IDirect3DDevice9 interface. This parameter defaults to NULL, but if you pass in the address of a device interface, this function will call the IDirect3DDevice9::SetViewport function to send your viewport parameters to the device. Note that we also set the **m_bProjDirty** variable to true. This means that the next time the application queries the state of the projection matrix, it will be rebuilt, taking the new aspect ratio of the viewport into account, as well as the new near and far plane values.

CCamera::GetProjMatrix

This function rebuilds the projection matrix and returns the result. The function only recalculates the projection matrix if the m_bProjDirty flag is set. It sets the m_bProjDirty flag to false after it is complete.

```
  const D3DXMATRIX& CCamera::GetProjMatrix()
  {
    // Only update matrix if something has changed
    if ( m_bProjDirty )
    {
        float fAspect = (float)m_Viewport.Width / (float)m_Viewport.Height;

        // Set the perspective projection matrix
        D3DXMatrixPerspectiveFovLH(&m_mtxProj, D3DXToRadian( m_fFOV / 2.0f ), fAspect,
                                   m_fNearClip, m_fFarClip );

        // Proj Matrix has been updated
        m_bProjDirty = false;
    }

    // Return the projection matrix.
    return m_mtxProj;
  }
```

The function is called by the CCamera::UpdateRenderProj matrix which is in turn called from the CGameApp class whenever the projection matrix of the camera needs to be updated. For example, when the window is resized, the viewport will need to be changed and the aspect ratio of this new window size calculated. So in the WM_SIZE handler, we would get the new window dimensions and call CCamera::SetViewport to record the data, and then call CCamera::UpdateRenderProj -- which would call the GetProjMatrix function -- to calculate the new projection matrix and set it as the device projection matrix.

CCamera::UpdateRenderProj / CCamera::UpdateRenderView

These functions are used to set the device view and projection matrices. Local matrices are rebuilt when their respective m_bProjDirty or m_bViewDirty Booleans are set to true.

```
void CCamera::UpdateRenderProj( LPDIRECT3DDEVICE9 pD3DDevice )
{
    if (!pD3DDevice) return;
    pD3DDevice->SetTransform( D3DTS_PROJECTION, &GetProjMatrix() );
}

void CCamera::UpdateRenderView( LPDIRECT3DDEVICE9 pD3DDevice )
{
    if (!pD3DDevice) return;
    pD3DDevice->SetTransform( D3DTS_VIEW, &GetViewMatrix() );
}
```

CCamera::GetViewMatrix

This function places the camera Right, Up, and Look vectors into columns 1, 2 and 3 of the view matrix respectively. It then places the inverted, view-space transformed position into the fourth row of the matrix. The view matrix is only rebuilt if its dirty flag is set. Just like the CPlayer class, we remember to perform vector regeneration at regular intervals to keep the axes perpendicular and unit length.

```
const D3DXMATRIX& CCamera::GetViewMatrix()
{
    // Only update matrix if something has changed
    if ( m_bViewDirty )
    {
        D3DXVec3Normalize( &m_vecLook, &m_vecLook );
        D3DXVec3Cross( &m_vecRight, &m_vecUp, &m_vecLook );
        D3DXVec3Normalize( &m_vecRight, &m_vecRight );
        D3DXVec3Cross( &m_vecUp, &m_vecLook, &m_vecRight );
        D3DXVec3Normalize( &m_vecUp, &m_vecUp );

        // Set view matrix values
        m_mtxView._11 = m_vecRight.x;m_mtxView._12 = m_vecUp.x;m_mtxView._13 = m_vecLook.x;
        m_mtxView._21 = m_vecRight.y;m_mtxView._22 = m_vecUp.y;m_mtxView._23 = m_vecLook.y;
        m_mtxView._31 = m_vecRight.z;m_mtxView._32 = m_vecUp.z;m_mtxView._33 = m_vecLook.z;
        m_mtxView._41 =- D3DXVec3Dot( &m_vecPos, &m_vecRight );
        m_mtxView._42 =- D3DXVec3Dot( &m_vecPos, &m_vecUp    );
        m_mtxView._43 =- D3DXVec3Dot( &m_vecPos, &m_vecLook  );

        // View Matrix has been updated
        m_bViewDirty = false;
    }
    // Return the view matrix.
    return m_mtxView;
}
```

The CCamfirstPerson Class

The first derived class we will examine will be the first person camera class. The class declaration can be found in CCamera.h.

```
class CCamfirstPerson : public CCamera
{
public:
     //Constructors
    CCamfirstPerson( const CCamera * pCamera );
    CCamfirstPerson();

     // Public Base Class Overrides
    CAMERA_MODE      GetCameraMode    ( ) const { return MODE_FPS; }
    void             Rotate           ( float x, float y, float z );
    void             SetCameraDetails ( const CCamera * pCamera );
};
```

There are two constructors, the first of which is a default constructor and the second of which is an overridden constructor that takes a pointer to a CCamera. Because we know that we are creating a first person camera in this constructor, we know exactly what information we need to extract from the passed camera in order to set the initial values. Also notice that the GetCameraMode function is implemented in the class declaration and simply returns MODE_FPS identifying that this is a first person camera object.

CCamfirstPerson:: CCamfirstPerson()

This constructor takes a CCamera object pointer so that it can clone its properties. It simply calls the CCamfirstPerson::SetCameraDetails functions to copy over the properties.

```
CCamfirstPerson::CCamfirstPerson( const CCamera * pCamera )
{
    // Update the camera from the camera passed
    SetCameraDetails( pCamera );
}
```

CCamfirstPerson::SetCameraDetails

The SetCameraDetails function copies properties from one camera to another. We did not hardcode the property copying code into the constructor so that the application can call SetCameraDetails to clone the settings of a camera at any time -- not just at camera class construction.

The first thing we do is check that a valid (non-NULL) pointer was passed. If this is not the case, we simply return. We can do this because the base class version of the function will have already been called, initialising the values to good defaults. We copy the position, clip planes, FOV, viewport, and volume information from the passed camera, as well as its Up, Look and Right vectors. If the camera we are cloning is a spacecraft camera then we need to flatten out the vectors. This is because the spacecraft mode is the only camera mode for which complete freedom of rotation is allowed about all three local axes. Finally, we make sure that we dirty both the projection matrix and the view matrix to force them to be rebuilt the next time they need to be sent to the device.

```
void CCamfirstPerson::SetCameraDetails( const CCamera * pCamera )
{
    // Validate Parameters
    if (!pCamera) return;

    // Reset / Clear all required values
    m_vecPos     = pCamera->GetPosition();
    m_vecRight   = pCamera->GetRight();
    m_vecLook    = pCamera->GetLook();
    m_vecUp      = pCamera->GetUp();
    m_fFOV       = pCamera->GetFOV();
    m_fNearClip  = pCamera->GetNearClip();
    m_fFarClip   = pCamera->GetFarClip();
```

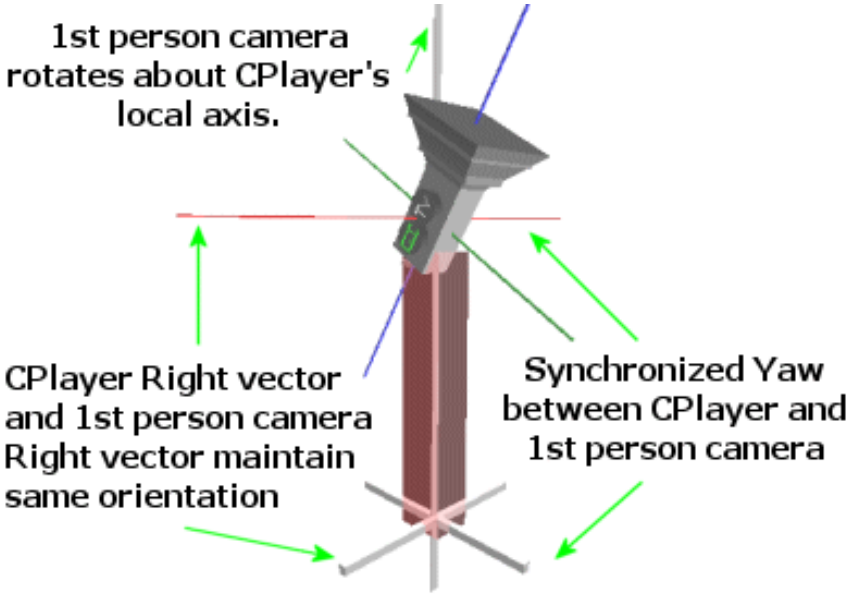
```
    m_Viewport   = pCamera->GetViewport();
    m_Volume     = pCamera->GetVolumeInfo();

    // If we are switching building from a spacecraft style cam
    if ( pCamera->GetCameraMode() == MODE_SPACECRAFT )
    {
        // Flatten out the vectors
        m_vecUp      = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );
        m_vecRight.y = 0.0f;
        m_vecLook.y  = 0.0f;

        // Finally, normalize them
        D3DXVec3Normalize( &m_vecRight, &m_vecRight );
        D3DXVec3Normalize( &m_vecLook, &m_vecLook );

    } // End if MODE_SPACECRAFT

    m_bViewDirty = true;
    m_bProjDirty = true;
}
```

CCamfirstPerson::Rotate

The Rotate function is an overridden virtual function. Recall that in CGameApp::ProcessInput we call CPlayer::Rotate in response the mouse being dragged with one or more buttons down. The CPlayer::Rotate function rotates the CPlayer Up, Look and Right vectors and then calls the attached camera's Rotate function.

```
void CCamfirstPerson::Rotate( float x, float y, float z )
{
    D3DXMATRIX mtxRotate;

    if ( x != 0 )
    {
        // Build rotation matrix
        D3DXMatrixRotationAxis( &mtxRotate, &m_vecRight, D3DXToRadian( x ) );

        // Update our vectors
        D3DXVec3TransformNormal( &m_vecLook, &m_vecLook, &mtxRotate );
        D3DXVec3TransformNormal( &m_vecUp, &m_vecUp, &mtxRotate );
    }
```

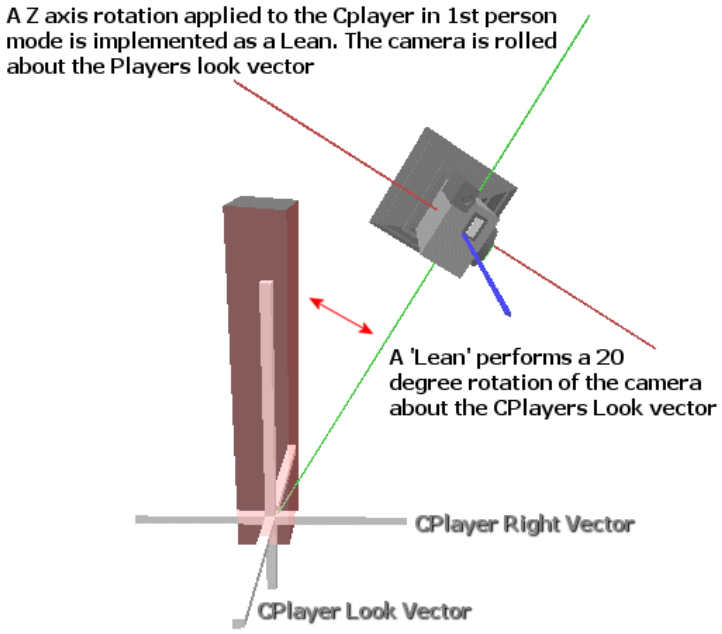
The first thing to check is whether an X axis rotation has been requested. Remember that in first person camera mode, we want the up/down mouse movements to rotate the camera about its own axis so the head can tilt up and down independent of the body. We build a rotation matrix that rotates vectors about the Right vector (the camera local X axis) and then rotates the Up and Look vectors around it by the specified angle.



Next we handle Y rotation if requested. In first person mode, the camera and the player yaw together, so the camera is rotated about the CPlayer Up vector.

```
    if ( y != 0 )
    {
        // Build rotation matrix
        D3DXMatrixRotationAxis( &mtxRotate, &m_pPlayer->GetUp(), D3DXToRadian( y ) );

        // Update our vectors
        D3DXVec3TransformNormal( &m_vecLook, &m_vecLook, &mtxRotate );
        D3DXVec3TransformNormal( &m_vecUp, &m_vecUp, &mtxRotate );
        D3DXVec3TransformNormal( &m_vecRight, &m_vecRight, &mtxRotate );

    } // End if Yaw
```

When a Z axis rotation has been requested, we need to implement a lean. Here we rotate the camera's Up, Look, and Right vectors as well as its position around the player Look vector:



Unlike other rotations, we need to rotate the axes of the camera and the camera world space position. Since all rotations are relative to the origin of the coordinate system, we must subtract the world space position of the player from the position of the camera, such that the player coordinate axes are situated at the origin. At this point we can rotate the position vector about the CPlayer Look vector (as shown in the above diagram) so that the camera is pivoted into a new position. We also rotate the camera local axes since these will change orientation. We use the function D3DXVec3TransformCoord to multiply a world space coordinate with a matrix, instead of the usual D3DXVec3TransformNormal:

```
    if ( z != 0 )
    {
        // Build rotation matrix
        D3DXMatrixRotationAxis( &mtxRotate, &m_pPlayer->GetLook(), D3DXToRadian( z ) );

        // Adjust camera position
        m_vecPos -= m_pPlayer->GetPosition();
        D3DXVec3TransformCoord ( &m_vecPos, &m_vecPos, &mtxRotate );
        m_vecPos += m_pPlayer->GetPosition();

        // Update our vectors
        D3DXVec3TransformNormal( &m_vecLook, &m_vecLook, &mtxRotate );
        D3DXVec3TransformNormal( &m_vecUp, &m_vecUp, &mtxRotate );
        D3DXVec3TransformNormal( &m_vecRight, &m_vecRight, &mtxRotate );
    }

    // Set view matrix as dirty
    m_bViewDirty = true;
}
```

Next we see the CCamfirstPerson::Rotate function in its entirety.

```
void CCamfirstPerson::Rotate( float x, float y, float z )
{
    D3DXMATRIX mtxRotate;
    if ( x != 0 )
    {
        // Build Rotation matrix
        D3DXMatrixRotationAxis( &mtxRotate, &m_vecRight, D3DXToRadian( x ) );

        // Update our vectors
        D3DXVec3TransformNormal( &m_vecLook, &m_vecLook, &mtxRotate );
        D3DXVec3TransformNormal( &m_vecUp, &m_vecUp, &mtxRotate );
        D3DXVec3TransformNormal( &m_vecRight, &m_vecRight, &mtxRotate );

    }

    if ( y != 0 )
    {
        // Build rotation matrix
        D3DXMatrixRotationAxis( &mtxRotate, &m_pPlayer->GetUp(), D3DXToRadian( y ) );

        // Update our vectors
        D3DXVec3TransformNormal( &m_vecLook, &m_vecLook, &mtxRotate );
        D3DXVec3TransformNormal( &m_vecUp, &m_vecUp, &mtxRotate );
        D3DXVec3TransformNormal( &m_vecRight, &m_vecRight, &mtxRotate );

    }

    if ( z != 0 )
    {
        // Build rotation matrix
        D3DXMatrixRotationAxis( &mtxRotate, &m_pPlayer->GetLook(), D3DXToRadian( z ) );

        // Adjust camera position
        m_vecPos -= m_pPlayer->GetPosition();
        D3DXVec3TransformCoord ( &m_vecPos, &m_vecPos, &mtxRotate );
        m_vecPos += m_pPlayer->GetPosition();

        // Update our vectors
        D3DXVec3TransformNormal( &m_vecLook, &m_vecLook, &mtxRotate );
        D3DXVec3TransformNormal( &m_vecUp, &m_vecUp, &mtxRotate );
        D3DXVec3TransformNormal( &m_vecRight, &m_vecRight, &mtxRotate );

    }

    // Set view matrix as dirty
    m_bViewDirty = true;
}
```

The CCamSpaceCraft Class

The declaration for the third person camera class is identical to the first person camera, and can also be found in CCamera.h. It overrides the same functions from the base class to provide custom rotations. Note that the GetCameraMode function returns MODE_SPACECRAFT.

```
class CCamSpaceCraft : public CCamera
{
public:
    // Constructors
    CCamSpaceCraft( const CCamera * pCamera );
    CCamSpaceCraft();

    // Public functions
    CAMERA_MODE         GetCameraMode    ( ) const { return MODE_SPACECRAFT; }
    void                Rotate           ( float x, float y, float z );
    void                SetCameraDetails ( const CCamera * pCamera );
};
```

The constructors are identical to that of the previous class, with a constructor that accepts a CCamera pointer and passes the request on to the SetCameraDetails function.

CCamSpaceCraft::SetCameraDetails

The SetCameraDetails function in this class is slightly different in that the spacecraft camera has total freedom of rotation. It does not have to flatten out any vectors as was the case with the CCamfirstPerson::SetCameraDetails function. This means that it simply copies the values straight into the class variables as shown below.

```
void CCamSpaceCraft::SetCameraDetails( const CCamera * pCamera )
{
    // Validate Parameters
    if (!pCamera) return;

    // Reset / Clear all required values
    m_vecPos    = pCamera->GetPosition();
    m_vecRight  = pCamera->GetRight();
    m_vecLook   = pCamera->GetLook();
    m_vecUp     = pCamera->GetUp();
    m_fFOV      = pCamera->GetFOV();
    m_fNearClip = pCamera->GetNearClip();
    m_fFarClip  = pCamera->GetFarClip();
    m_Viewport  = pCamera->GetViewport();
    m_Volume    = pCamera->GetVolumeInfo();

    // Rebuild both matrices
    m_bViewDirty = true;
    m_bProjDirty = true;
}
```

CCamSpaceCraft::Rotate

Unlike the first person camera mode where the camera can have rotations independent from the CPlayer (such as pitching the camera up and down about its own axes) the spacecraft camera has its rotations synchronized with the CPlayer rotation. If the player rotates about his Y axis, then the spacecraft camera also rotates about the CPlayer Y axis. In our application, we set the camera offset to

zero when the change is made to spacecraft mode so that the camera is always in exactly the same position as the player. Because rotations are paralleled by both classes, the Up, Right, and Look vectors of both remain identical throughout. You can think of the CPlayer object as a spaceship with total freedom of rotation, and the camera as the pilot in the cockpit who rotates when the space craft rotates. For example, you may decide that your space craft is a big mother ship and the bridge of the ship is offset 50 units from the player origin. That is why this rotation function always rotates the camera about the CPlayer axes. Doing this makes sure that rotations are handled correctly even if there is a camera offset being used. The only difference in the code is that we must subtract the player position from the camera position so that the rotation happens relative to the origin of the coordinate system. Once the position has been rotated, we add the CPlayer positions back on to the camera position to restore it to its new position in world space.

```
void CCamSpaceCraft::Rotate( float x, float y, float z )
{
    D3DXMATRIX mtxRotate;

    if ( x != 0 ){
        // Build rotation matrix about players X axis
        D3DXMatrixRotationAxis( &mtxRotate, &m_pPlayer->GetRight(), D3DXToRadian( x ) );
        D3DXVec3TransformNormal( &m_vecLook, &m_vecLook, &mtxRotate );
        D3DXVec3TransformNormal( &m_vecUp, &m_vecUp, &mtxRotate );

        // Rotate about player
        m_vecPos -= m_pPlayer->GetPosition();
        D3DXVec3TransformCoord( &m_vecPos, &m_vecPos, &mtxRotate );
        m_vecPos += m_pPlayer->GetPosition();
    }

    if ( y != 0 ){
        // Build rotation matrix
        D3DXMatrixRotationAxis( &mtxRotate, &m_pPlayer->GetUp(), D3DXToRadian( y ) );
        D3DXVec3TransformNormal( &m_vecLook, &m_vecLook, &mtxRotate );
        D3DXVec3TransformNormal( &m_vecRight, &m_vecRight, &mtxRotate );

        // Adjust position
        m_vecPos -= m_pPlayer->GetPosition();
        D3DXVec3TransformCoord( &m_vecPos, &m_vecPos, &mtxRotate );
        m_vecPos += m_pPlayer->GetPosition();
    }

    if ( z != 0 ){
        // Build rotation matrix
        D3DXMatrixRotationAxis( &mtxRotate, &m_pPlayer->GetLook(), D3DXToRadian( z ) );
        D3DXVec3TransformNormal( &m_vecUp, &m_vecUp, &mtxRotate );
        D3DXVec3TransformNormal( &m_vecRight, &m_vecRight, &mtxRotate );

        // Adjust position
        m_vecPos -= m_pPlayer->GetPosition();
        D3DXVec3TransformCoord( &m_vecPos, &m_vecPos, &mtxRotate );
        m_vecPos += m_pPlayer->GetPosition();
    }

    // Set view matrix as dirty
    m_bViewDirty = true;
}
```

The CCamthirdPerson Class

This class is implemented quite differently than the previous two. First we notice that the Move and Rotate functions are overridden but have no function bodies. Any calls from the CPlayer to move or rotate the third person camera are ignored. We have also overridden the CCamera::Update function. Recall that in the CPlayer::Update function, CPlayer::Move is called to update the player position using the current velocity vector. This function then passes the move request on to the camera. In first person and spacecraft camera modes, this move request moves the camera along the velocity vector to its new position. In this class however, it does nothing. The next thing that the CPlayer::Update function does after the CPlayer has been moved to its new position, is call the CCamera::Update function. This function does nothing in first person and spacecraft camera mode, but in this class it is used to move the camera to a new position that follows the CPlayer object.

```
class CCamthirdPerson : public CCamera
{
public:

    // Constructors
    CCamthirdPerson( const CCamera * pCamera );
    CCamthirdPerson();

    // Public Functions for This Class.
    CAMERA_MODE         GetCameraMode    ( ) const { return MODE_THIRDPERSON; }
    void                Move             ( const D3DXVECTOR3& vecShift ) {};
    void                Rotate           ( float x, float y, float z )   {};
    void                Update           ( float TimeScale, float Lag );
    void                SetCameraDetails ( const CCamera * pCamera );
    void                SetLookAt        ( const D3DXVECTOR3& vecLookAt );
};
```

We will not look at the code to the **SetCameraDetails** function since it is identical to that of its CCamfirstPerson equivalent. It simply copies over the details of the passed CCamera and flattens out the vectors on to the XZ plane if the camera passed was previously in spacecraft mode. Remember that the CPlayer in third person mode is limited to rotation about its Y axis only (Yaw).

CCamthirdPerson::Update

The CCamthirdPerson::Update function is the core of this class. It is called every frame of the game (because it is called from CPlayer::Update which is called every frame) and makes sure that the camera follows the player. It uses the camera lag setting to smooth any rotations that occur.

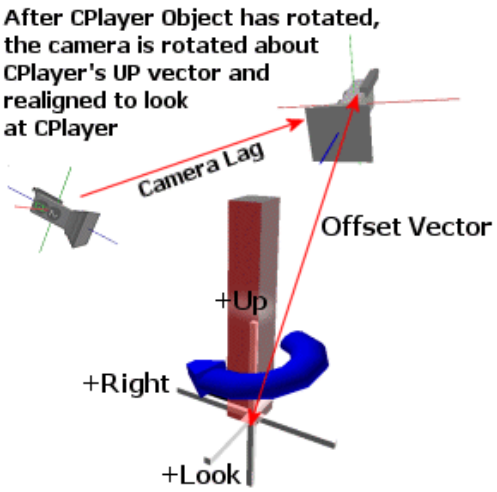
When this function is called from the CPlayer::Update function, it is passed the elapsed time since the last frame as well as the camera lag setting (previously set with a call to CPlayer::SetCamLag). This controls how quickly the camera catches up to changes in player orientation and position. We will multiply the elapsed time by the reciprocal of the lag value and use this as a scaling value for this frame. Larger lags result in slower camera movement along its movement vector in a single update.

```
void CCamthirdPerson::Update( float TimeScale, float Lag )
{
    D3DXMATRIX  mtxRotate;
    D3DXVECTOR3 vecOffset, vecPosition, vecDir;

    float fTimeScale = 1.0f, Length = 0.0f;
    if ( Lag != 0.0f ) fTimeScale = TimeScale * (1.0f / Lag);
```

Now that we have the time scale, we need to take the camera offset vector (set by SetCamOffset) and transform it so that it is relative to the player. Why do we do this? Let us imagine that we initially set the camera offset vector to (0, 0,-10) to indicate that we want the camera to be 10 units behind the player. We will want this to be true regardless of the way the player is oriented. We know that if the player has a look vector of (1, 0, 0) they are looking down the world X axis. In this instance, if the player were positioned at the origin, 10 units 'behind' the player would actually be (-10, 0, 0) since the back side of the player is facing down the -X axis. Therefore, we need to take the offset and convert it from a player space offset vector into a world space offset vector. To do this, we build a temporary rotation matrix (without the translation vector in the fourth row) for the player and multiply the offset vector by this matrix. It is rotated by the CPlayer local axes so that the offset is now a world space offset. All we have to do is add this world space offset to the player world space position and we have the world space position of the point where the camera belongs.

```
    // Rotate our offset vector to its position behind the player
    D3DXMatrixIdentity( &mtxRotate );
    D3DXVECTOR3 vecRight = m_pPlayer->GetRight(), vecUp = m_pPlayer->GetUp(),
                vecLook = m_pPlayer->GetLook();
    mtxRotate._11 = vecRight.x; mtxRotate._21 = vecUp.x; mtxRotate._31 = vecLook.x;
    mtxRotate._12 = vecRight.y; mtxRotate._22 = vecUp.y; mtxRotate._32 = vecLook.y;
    mtxRotate._13 = vecRight.z; mtxRotate._23 = vecUp.z; mtxRotate._33 = vecLook.z;

    // Calculate our rotated offset vector
    D3DXVec3TransformCoord( &vecOffset, &m_pPlayer->GetCamOffset(), &mtxRotate );

    // vecOffset now contains information to calculate where our camera position SHOULD be.
    vecPosition = m_pPlayer->GetPosition() + vecOffset;
```

If we were not using lag, then we could immediately update the camera position to this newly calculated position. However, such a transition would appear abrupt and we would prefer that our camera gently glides into place over the next few frames. So we will calculate a direction vector from the camera current position to the newly calculated position and move the camera along this vector instead. The distance we move along this vector is dependant on the time scale calculated above. The next image makes clear our objectives.




```
    vecDir = vecPosition - m_vecPos;
    Length = D3DXVec3Length( &vecDir );
    D3DXVec3Normalize( &vecDir, &vecDir );

    // Move based on camera lag
    float Distance = Length * fTimeScale;
    if ( Distance > Length ) Distance = Length;

    // If we only have a short way to travel, move all the way
    if ( Length < 0.01f ) Distance = Length;

    // Update our camera
    if ( Distance > 0 )
    {
        m_vecPos += vecDir * Distance;

        // Ensure our camera is looking at the axis origin
        SetLookAt( m_pPlayer->GetPosition() );

        // Our view matrix parameters have been update
        m_bViewDirty = true;
    }
}
```

We calculate the vector from the current position to the desired position and record the length of this vector so that we know how far we have to travel in that direction. We then normalize the vector so that it is unit length. Next we scale the distance by the time scale to get the distance we can travel in this single update. If the distance to the desired position is very small, we immediately assign the desired position to the camera position. Otherwise, we scale the new direction vector by the time scale to produce a velocity vector for this update. This vector is then added to the camera position.

CCamthirdPerson::SetLookAt

In third person mode, we will make sure that the camera always faces the player. This function adjusts the Look, Up, and Right vectors so that the camera points in the correct direction. Rather than calculate the new vectors ourselves, we can use the D3DXMatrixLookAtLH function to build the matrix for us. We can then extract the new vectors from the matrix directly into the camera member variables.

```
void CCamthirdPerson::SetLookAt( const D3DXVECTOR3& vecLookAt )
{
    D3DXMATRIX Matrix;

    // Generate a look at matrix
    D3DXMatrixLookAtLH( &Matrix, &m_vecPos, &vecLookAt, &m_pPlayer->GetUp() );

    // Extract the vectors
    m_vecRight = D3DXVECTOR3( Matrix._11, Matrix._21, Matrix._31 );
    m_vecUp    = D3DXVECTOR3( Matrix._12, Matrix._22, Matrix._32 );
    m_vecLook  = D3DXVECTOR3( Matrix._13, Matrix._23, Matrix._33 );

    // Set view matrix as dirty
    m_bViewDirty = true;
}
```

The function is passed the position in world space we wish to look at. This value, along with the camera current position and the player Up vector, is passed into the D3DX function to build the matrix. What we are doing here is building a matrix for an object situated at m_vecPos (camera current position) looking at the player position (vecLookAt) in such a way that its Up vector is aligned with the player Up vector. Keep in mind that the D3DXMatrixLookAt function builds a view matrix, which is an inverse matrix. This is why we extract the vectors from its columns and not its rows.

CTerrain Revisited

All that is left to do is discuss the functions that handle the player and camera collision detection against the terrain. As we have already mentioned, the CTerrain class provides two static call-back functions which are added to the CPlayer call-back function arrays. The first function we will look at is the CTerrain::UpdatePlayer function. It is called by CPlayer::Update to allow CTerrain to modify the position of the CPlayer object when it intersects the terrain.

CTerrain::UpdatePlayer (*static*)

```
void CTerrain::UpdatePlayer( LPVOID pContext, CPlayer * pPlayer, float TimeScale )
{
    // Validate Parameters
    if ( !pContext || !pPlayer ) return;

    VOLUME_INFO Volume    = pPlayer->GetVolumeInfo();
```

```
    D3DXVECTOR3 Position  = pPlayer->GetPosition();
    D3DXVECTOR3 Velocity  = pPlayer->GetVelocity();
    bool        ReverseQuad = false;
```

First we store the values we will need to test for terrain collision. In a moment, we will call CTerrain::GetHeight to retrieve the current height of the terrain at the position of the player. Essentially, GetHeight uses the current X and Z position of the player to find the four pixels in the height map which define the quad the player is currently standing on. It will then interpolate the height values between these four corner points to find the actual height of the terrain -- which may be some point in between those four points. In order to do this, the GetHeight function needs to know whether we are on an even or odd row of the terrain. In Chapter Three we saw that the terrain is represented as a triangle strip. The first row is rendered from left to right, the second row is rendered right to left, the third row is rendered left to right, and so on. We determine odd or even by dividing the player world space Z position by CTerrain::m_vecScale. This converts the Z coordinate into a height map space row. If we have a 10x10 height map and we have a terrain scale factor of 10, then the terrain will be 100x100 in world space. If the Z coordinate of our player was 25:

CPlayer.Z = 25
CTerrain.m_vecScale = 10;
Row = 25/10 = 2 (We are on the third row , so this is an odd row)

Here is the code to the function that calculates this.

```
    // Determine which row we are on
    int PosZ = (int)(Position.z / ((CTerrain*)pContext)->m_vecScale.z);
    if ( (PosZ % 2) != 0 ) ReverseQuad = true;
```

Here we call CTerrain:GetHeight to retrieve the height of the terrain under the player. We make sure to pass in the Boolean we just calculated so that the function knows whether we are on an odd or even row.

```
  // Retrieve the height of the terrain at this position
  float vy = Volume.Min.y;
  float fHeight = ((CTerrain*)pContext)->GetHeight(Position.x,Position.z,ReverseQuad) - vy;
```

We pass the X and Z position of the CPlayer to index into the height map and calculate the terrain height at a specific point. Once this height is returned, we subtract the world space Y position of the bounding volume minimum Y point. We do this because we wish to know if the terrain intersects the player bounding volume. Note that the lowest point in the bounding volume may be lower than the actual position of the player himself. For example, we may have a defined a bounding box where the player position is at the center. So we need to test that the bottom of the bounding box does not intersect the terrain.

Finally, we check to see if the world space position of the player is lower than the height of the terrain at that point. If so, the height of the player is modified so that its new position is exactly the height of

the terrain. This means if the bounding volume was intersecting the terrain, it will be moved upwards so that the bounding volume sits on the terrain at the correct height.

```
    // Determine if the position is lower than the height at this position
    if ( Position.y < fHeight )
    {
        // Update camera details
        Velocity.y = 0;
        Position.y = fHeight;

        // Update the camera
        pPlayer->SetVelocity( Velocity );
        pPlayer->SetPosition( Position );

    } // End if colliding
}
```

Here is the CTerrain::UpdatePlayer function in its entirety:

```
void CTerrain::UpdatePlayer( LPVOID pContext, CPlayer * pPlayer, float TimeScale )
{
    // Validate Parameters
    if ( !pContext || !pPlayer ) return;

    VOLUME_INFO Volume   = pPlayer->GetVolumeInfo();
    D3DXVECTOR3 Position = pPlayer->GetPosition();
    D3DXVECTOR3 Velocity = pPlayer->GetVelocity();
    bool        ReverseQuad = false;

    // Determine which row we are on
    int PosZ = (int)(Position.z / ((CTerrain*)pContext)->m_vecScale.z);
    if ( (PosZ % 2) != 0 ) ReverseQuad = true;

    // Retrieve the height of the terrain at this position
    float vy = Volume.Min.y;
    float fHeight = ((CTerrain*)pContext)->GetHeight(Position.x,
                                                     Position.z,
                                                     ReverseQuad) - vy;

    // Determine if the position is lower than the height at this position
    if ( Position.y < fHeight )
    {
        // Update camera details
        Velocity.y = 0;
        Position.y = fHeight;

        // Update the camera
        pPlayer->SetVelocity( Velocity );
        pPlayer->SetPosition( Position );
    }
}
```

CTerrain::UpdateCamera

CPlayer::UpdateCamera is called by the CPlayer::Update function every frame to give CTerrain a chance to modify the position of the camera if it has moved into an illegal position. This function is very similar to the UpdatePlayer function so we will show it in its entirety with only a brief description.

```
void CTerrain::UpdateCamera( LPVOID pContext, CCamera * pCamera, float TimeScale )
{
    // Validate Requirements
    if (!pContext || !pCamera ) return;
    if ( pCamera->GetCameraMode() != CCamera::MODE_THIRDPERSON ) return;

    VOLUME_INFO Volume   = pCamera->GetVolumeInfo();
    D3DXVECTOR3 Position = pCamera->GetPosition();
    bool        ReverseQuad = false;

    // Determine which row we are on
    ULONG PosZ = (ULONG)(Position.z / ((CTerrain*)pContext)->m_vecScale.z);
    if ( (PosZ % 2) != 0 ) ReverseQuad = true; else ReverseQuad = false;

    float vy = Volume.Min.y;
    float fHeight = ((CTerrain*)pContext)->GetHeight(Position.x,
                                                     Position.z,
                                                     ReverseQuad) - vy;

    // Determine if the position is lower than the height at this position
    if ( Position.y < fHeight )
    {
        // Update camera details
        Position.y = fHeight;
        pCamera->SetPosition( Position );

    } // End if colliding

    // Retrieve the player at which the camera is looking
    CPlayer * pPlayer = pCamera->GetPlayer();
    if (!pPlayer) return;

    // We have updated the position of either our player or camera
    // We must now instruct the camera to look at the players position
    ((CCamthirdPerson*)pCamera)->SetLookAt(  pPlayer->GetPosition() );
}
```

We start by retrieving the information from the passed camera and then calculate whether the camera position is on an odd or even row in height map space. Next we call CTerrain::GetHeight to retrieve the current height of the terrain underneath the camera. Note that this function returns immediately if the camera is not a third person camera. Only in third person mode does the camera really have a chance to intersect the terrain of its own accord. In first person mode for example, the camera is fixed at a specified offset from the terrain. If the player is embedded in the terrain and corrected by the UpdatePlayer function, the camera position will also be adjusted as a result.

We test whether the camera Y coordinate is lower than the terrain at that height and if so, the position is adjusted to the new height. This is very important in third person mode since the camera is trying to follow the player and as such, its path might take it straight through the landscape. This code ensures that even when the player is on the other side of a mountain, the camera will gracefully drift over the top of the mountain to catch up, instead of flying straight through it.

Finally, we must make sure that the camera is still looking directly at the player at all times. If this function had to correct the camera position by a significant amount, it is entirely possible that the camera would be moved such that it no longer directly faces the player. Therefore, when we correct the camera position, we also call the CCamthirdPerson::SetLookAt function to make sure the Look vector is adjusted appropriately.

CTerrain::GetHeight

This function uses the world space X and Z coordinates to determine the exact height of the terrain at that point. It does this by first dividing the world space X and Z coordinates by the terrain scale vector so that the coordinate pair is in image space. Now those values will describe a pixel in the height map – i.e. a height value. We can use this image space point to calculate the three neighbouring image space pixel heights. This gives us four pixels in the height map describing the heights of the quad corner points. This is the quad that the world space point is positioned over.

```
float CTerrain::GetHeight( float x, float z, bool ReverseQuad )
{
    float fTopLeft, fTopRight, fBottomLeft, fBottomRight;

    // Adjust Input Values
    x = x / m_vecScale.x;
    z = z / m_vecScale.z;

    // Make sure we are not OOB
    if ( x < 0.0f || z < 0.0f || x >= m_nHeightMapWidth || z >= m_nHeightMapHeight )
        return 0.0f;

    // First retrieve the Heightmap Points
    int ix = (int)x;
    int iz = (int)z;
    // Calculate the remainder (percent across quad)
    float fPercentX = x - ((float)ix);
    float fPercentZ = z - ((float)iz);
```

We divide the world space coordinate pair by the scale vector to produce an image space value. Thus, if the terrain is 100x100 and has a scale vector of 10 and we pass in coordinates (72, 28):

flImageSpaceX = 72/10 = 7.2
flImageSpaceZ = 28/10 = 2.8 (Remember, the Z value is really the Y coordinate in image space.)

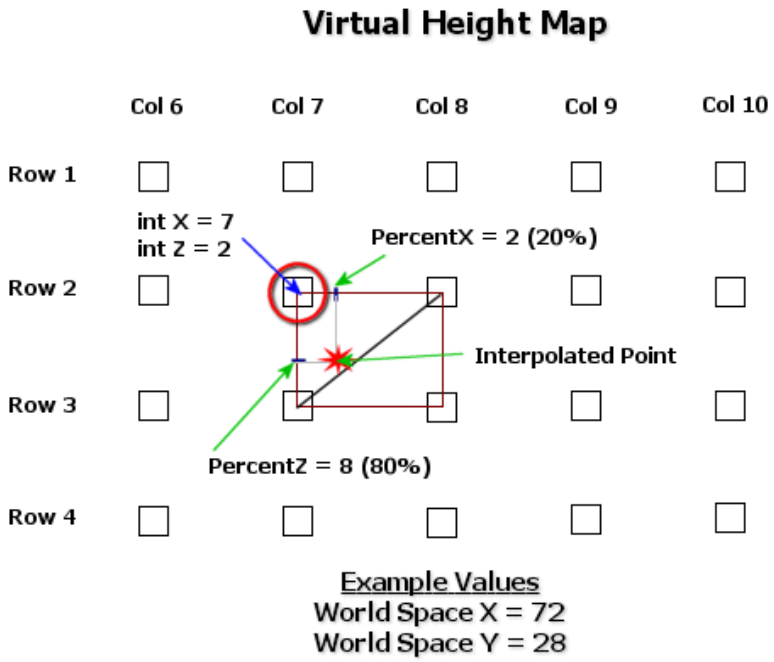
The result indicates that the point is between the 7th and 8th pixel horizontally in the image map and between the 2nd and 3rd pixels vertically down the image map.

iImageSpaceX = 7
iImageSpaceY = 2
We now have an image space coordinate that describes one of the points making up the quad that the world space point is currently over. We will use the remainder as a percentage between 0.0 and 1.0 to describe how close this point is to each point in the quad. We subtract the integer from the float so that we are left with the remainders shown below.

PercentX = 2 (This means the world space position is between pixel 7 and 8. If you were to draw a line horizontally between pixels 7 and 8, the position would be 20% along this line.)

PercentZ = 8 (This means that the position is between rows 8 and 9 in the image. If you were to draw a vertical line from row 8 to row 9, the position would be 80% along this line. In other words the point is nearer to row 9.)

The next image shows how we will use these percentage values to determine a virtual location between four neighbouring pixels in the height map.

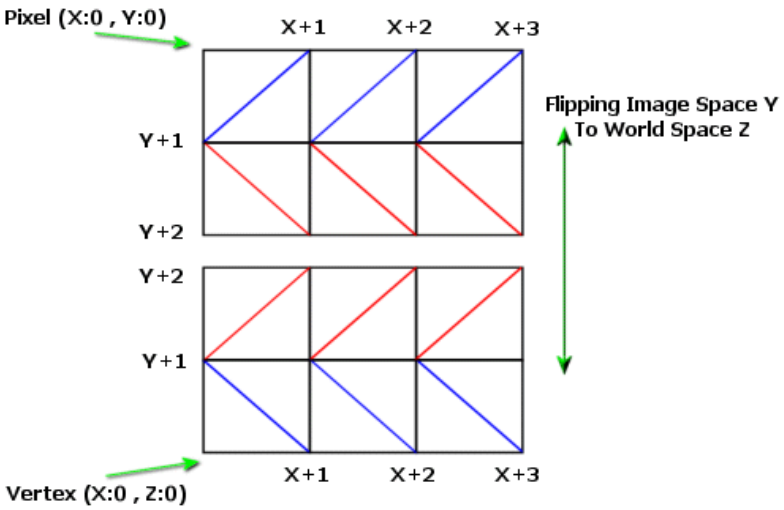


We know that the image map cannot use fractional coordinates because its pixels are at discrete locations. For example, there is no way for us to access pixel (7.2, 2.8) in an image. But the above image shows that if we imagine a virtual height map such that this is the case, we find that the pixels are spaced out much like the terrain vertices after they have been scaled by the scaling vector. We can see that the coordinate (72, 28) is inside the quad represented by pixels/vertices (7, 2), (8, 2), (8, 3) and (7, 3). When we imagine the image pixels in the height map being spaced out like this, we can see that it actually mirrors the way the terrain vertices were created. They were originally assigned pixel

positions (right next to each other in the height map with no gaps in between them) and then the vertex positions were scaled and the vertices were separated. Every four vertices defined two triangles (a quad) on the terrain. We cannot simply extract a height value from a pixel in the height map since the world space position passed in may be *between* vertices and thus between the integer height values in the height map. So we calculated an offset (7, 2) into the height map to give us one of the quad positions. We then use the remainder of each coordinate to tell us the position between adjacent pixels in the height map. In the above image, the coordinate (7.2, 2.8) describes a virtual location between rows 2 and 3 and between columns 7 and 8 (marked as a red star). Retrieving the height values of the four integer locations of the quad in the height map allow us to interpolate the actual height of the position that falls between those four points.

Remember that the image space points give us the unscaled height of each vertex in the quad that we are over. This means that all we have to do is multiply the four values by the scale vector (just as we did when we built the terrain vertices initially) and we get the four height values for the four vertices making up the quad in the terrain. We will refer to these points as TopLeft, TopRight, BottomLeft, and BottomRight. Because the quad is actually made up of two triangles which may belong to different planes, we first retrieve the two corner points of the dividing edge that splits the two triangles. We use will use these edge points later to determine which triangle the point is in. One thing you have to remember when looking at the following code is that the dividing edge faces a different way depending on whether we are on an odd or even row of the terrain. This is the result of the degenerate triangles used to move up to the next row in the strip. Also keep in mind that if you were to position the camera high above and look down on the terrain, the Y direction of the image is flipped when it is used as the Z component of each vertex. This is because in image space, the first pixel is at the top left corner of the screen with increasing X values to the right. In world space, pixel zero is mapped to vertex zero which is at the bottom left corner of the terrain. When we access image data, we must take this Y-Z flip into account and remember that the dividing edges are actually in opposite directions.

The following picture shows two rows of quads. The top image shows how the quads are built if we were to draw the quads in image space. The bottom image shows what they look like in world space if we were looking down on the terrain. Notice how the X coordinates increase in the same direction in both image space and world space, but the Y coordinate increases down the screen in image space while the Z coordinate of the vertex increases up the screen.



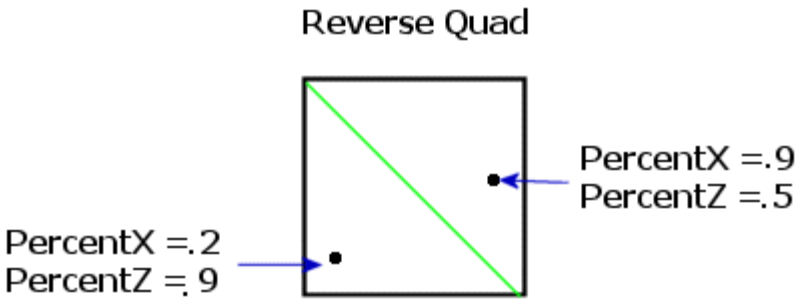
We can see from this image that on even rows in image space, the dividing edge of the two triangles making up a quad goes from top right to bottom left. On odd rows, the dividing edge goes from top left to top right. This is why we needed to calculate whether the player or camera position was on an odd or even row in the calling function.

With this knowledge, we calculate the dividing edge height points first. After we do that we need to calculate the next two points such that the four points make a planar quad. Since the two triangles making up the quad may not be planar, we do a test between the X percentage and the Z percentage to figure out which triangle we are in. Once we have the triangle, we have three planar points. With these points, we can then calculate the final point of the planar quad. Notice in the following code that this has to be done differently depending on whether we are on an odd or even row (reversed quad or not).

```
    if ( ReverseQuad )
    {
        // First retrieve the height of each point in the dividing edge
        fTopLeft     = (float)m_pHeightMap[ix + iz * m_nHeightMapWidth] * m_vecScale.y;
        fBottomRight = (float)m_pHeightMap[(ix + 1) + (iz + 1) * \
                                           m_nHeightMapWidth] * m_vecScale.y;
```

We now have the dividing edge if this is a reverse quad (a quad from an odd row). The variables were multiplied by the terrain scale vector (Y component only) so that they now contain the world space vertex heights of the two vertices making up the dividing edge of the quad.

Now we need to figure out which triangle of the quad we are in. Fortunately, because we are working with height map coordinates, the quad is still a perfect square. This would not be the case if we were dealing with world space X and Z coordinates -- where the scale vector of the terrain might have scaled the positions more along the X axis than the Z axis. This means that the line forming the dividing edge is a perfect diagonal. Testing whether any point is within the top right triangle or the bottom left triangle is a simple case of comparing the X coordinate to the Z coordinate. If the X coordinate is smaller than the Z coordinate, then we are in the top right triangle, otherwise we are in the bottom left triangle as the next image demonstrates:



Once we know which triangle we are in, we get the final point of the triangle and construct a point that is on the triangle plane to build a planar quad. The following code does this depending on whether the point is in the left or right triangle:

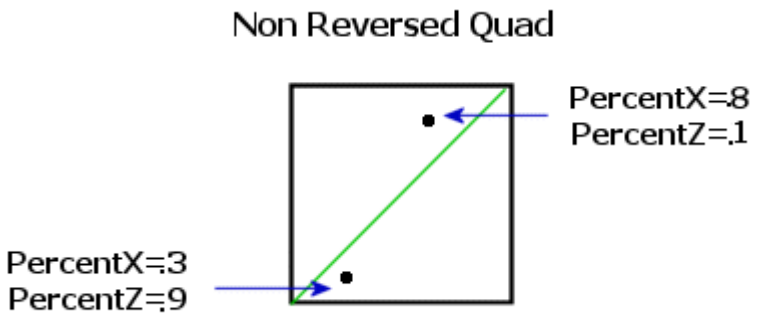
```
        // Which triangle of the quad are we in ?
        if ( fPercentX < fPercentZ )
        {
             fBottomLeft = (float)m_pHeightMap[ix + (iz + 1) * \
                                   m_nHeightMapWidth] * m_vecScale.y;
                               fTopRight   = fTopLeft + (fBottomRight - fBottomLeft);
        } // End if Left Triangle
```

The fBottomLeft variable contains the height of the bottom left vertex in the quad. We use the three triangle points to create a top right vertex height which is co-planar with the other three triangles points. This top right vertex may not be the height of that vertex in the terrain, but it does not matter. We already know that we are in the left triangle, so we just need a planar quad to interpolate the actual height value.

If the X coordinate is greater than the Z coordinate, then we are in the right triangle. This means we need the top right height value to complete our triangle points, and we need to build the bottom left height value such that we have a planar quad. Remember, the quad may not actually be planar in our actual terrain, but the triangle is the correct world space triangle. The fourth point is needed so that we have a temporary planar quad to interpolate the height value.

```
        else
        {
            fTopRight   = (float)m_pHeightMap[(ix + 1) + iz * \
                                  m_nHeightMapWidth] * m_vecScale.y;
            fBottomLeft = fTopLeft + (fBottomRight - fTopRight);
        } // End if Right Triangle
    } // End if Quad is reversed
```

At this point we have a planar quad if we are processing a reversed quad. If this is not a reversed quad and we are processing a quad from an even row, then we need to take into account the fact that the dividing edge will be facing the other way as the image below shows:



In this case the edge is between the top right and bottom left points in the height map:

```
    else
    {
        // First retrieve the height of each point in the dividing edge
        fTopRight   = (float)m_pHeightMap[(ix + 1) + iz * \
                                          m_nHeightMapWidth] * m_vecScale.y;
        fBottomLeft = (float)m_pHeightMap[ix + (iz + 1) * \
                                          m_nHeightMapWidth] * m_vecScale.y;
```

Because the line is facing in the other direction, we need to modify the test to determine which triangle the point is in. In this case we have to test if fPercentX is smaller than 1.0-fPercentZ. For example, if we take the point (0.8, 0.2) and use that in our example, we can see that 0.8 is smaller than 1.0 - 0.1 = 0.9. So we are in the left triangle in that case. We can also see that for the second point above, 0.3 is not smaller than 1.0 - 0.9 = 0.1 so we must be in the right triangle. Depending on which triangle we are in, we extract the third point of the triangle from the height map and build the fourth point for the planar quad. Here is the code:

```
        // Calculate which triangle of the quad are we in ?
        if ( fPercentX < (1.0f - fPercentZ))
        {
            fTopLeft = (float)m_pHeightMap[ix + iz * m_nHeightMapWidth] * m_vecScale.y;
            fBottomRight = fBottomLeft + (fTopRight - fTopLeft);

        } // End if Left Triangle
        else
        {
            fBottomRight = (float)m_pHeightMap[(ix + 1) + (iz + 1)* m_nHeightMapWidth];
            fBottomRight *= m_vecScale.y;
            fTopLeft = fTopRight + (fBottomLeft - fBottomRight);

        } // End if Right Triangle

    } // End if Quad is not reversed
```

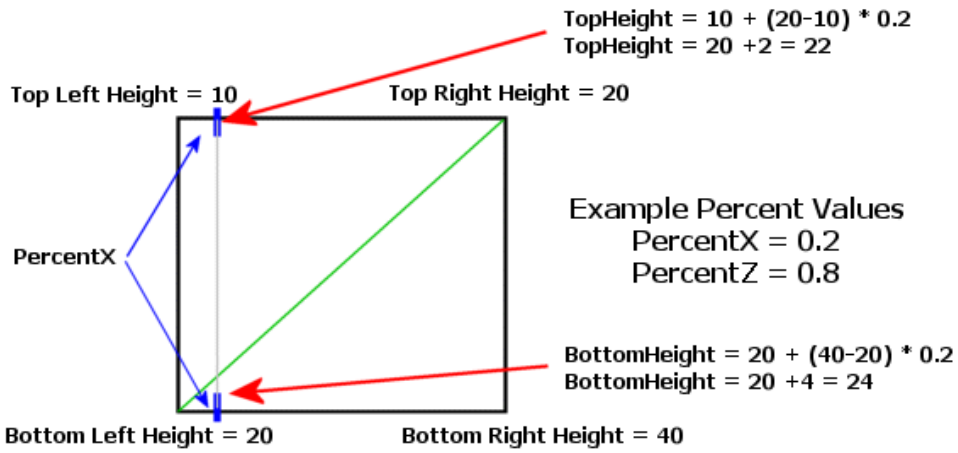
At this point we have a planar quad of height values. We multiply the top right height value by the fPercentX fraction and add this to the top left height value. This creates an edge between the top left and right points in the quad. We interpolate along the edge to get the height of that edge at the correct horizontal position.

```
    // Calculate the height interpolated across the top and bottom edges
    float fTopHeight    = fTopLeft    + ((fTopRight - fTopLeft) * fPercentX );
```

We now do exactly the same with the bottom edge of the quad as shown below.

```
    float fBottomHeight = fBottomLeft + ((fBottomRight - fBottomLeft) * fPercentX );
```

So we now have two height values: one on the top edge and one on the bottom edge. The following image shows how the top and bottom height values would be calculated using some example values for both the four vertex height values and using a PercentX of 0.2:

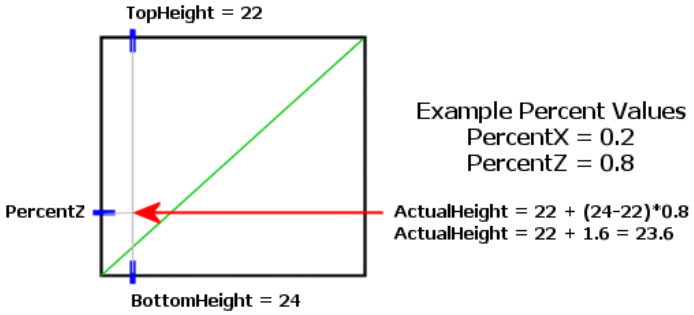


As you can see, the PercentX value tells us how far we have to interpolate the height value along the top and bottom edges. In the above example, the final height values for the top and bottom edges are 22 and 24 for the top and bottom edges respectively.

The previous image should give you a hint as to how we calculate the final height. We interpolate along the line formed from the top and bottom height values. The interpolation distance is what PercentZ is used for.

```
    // Calculate the resulting height interpolated between the two heights
    return fTopHeight + ((fBottomHeight - fTopHeight) * fPercentZ );
}
```

The next image shows how the last line of code works to calculate the actual height value:




```
float CTerrain::GetHeight( float x, float z, bool ReverseQuad )
{
    float fTopLeft, fTopRight, fBottomLeft, fBottomRight;

    // Adjust Input Values
    x = x / m_vecScale.x;
    z = z / m_vecScale.z;

    // Make sure we are not OOB
    if ( x < 0.0f || z < 0.0f || x >= m_nHeightMapWidth || z >= m_nHeightMapHeight )
         return 0.0f;

    // First retrieve the Heightmap Points
    int ix = (int)x;
    int iz = (int)z;

    // Calculate the remainder (percent across quad)
    float fPercentX = x - ((float)ix);
    float fPercentZ = z - ((float)iz);

    if ( ReverseQuad )
    {
        // First retrieve the height of each point in the dividing edge
        fTopLeft     = (float)m_pHeightMap[ix + iz * m_nHeightMapWidth] * m_vecScale.y;
        fBottomRight = (float)m_pHeightMap[(ix + 1) +(iz + 1) * m_nHeightMapWidth] *
                       m_vecScale.y;

        // Which triangle of the quad are we in ?
        if ( fPercentX < fPercentZ )
        {
            fBottomLeft = (float)m_pHeightMap[ix + (iz + 1) * m_nHeightMapWidth] *
                          m_vecScale.y;
            fTopRight = fTopLeft + (fBottomRight - fBottomLeft);

        } // End if Left Triangle
        else
        {
            fTopRight = (float)m_pHeightMap[(ix + 1) + iz * m_nHeightMapWidth] *
                        m_vecScale.y;
            fBottomLeft = fTopLeft + (fBottomRight - fTopRight);

        } // End if Right Triangle

    } // End if Quad is reversed
    else
    {
```

```
        // First retrieve the height of each point in the dividing edge
        fTopRight   = (float)m_pHeightMap[(ix + 1) + iz * m_nHeightMapWidth] *
                      m_vecScale.y;
        fBottomLeft = (float)m_pHeightMap[ix + (iz + 1) * m_nHeightMapWidth] *
                      m_vecScale.y;

        // Calculate which triangle of the quad are we in ?
        if ( fPercentX < (1.0f - fPercentZ))
        {
            fTopLeft = (float)m_pHeightMap[ix + iz * m_nHeightMapWidth] * m_vecScale.y;
            fBottomRight = fBottomLeft + (fTopRight - fTopLeft);

        } // End if Left Triangle
        else
        {
            fBottomRight = (float)m_pHeightMap[(ix + 1) + (iz + 1) * m_nHeightMapWidth] *
                           m_vecScale.y;
            fTopLeft = fTopRight + (fBottomLeft - fBottomRight);

        } // End if Right Triangle

    } // End if Quad is not reversed

    // Calculate the height interpolated across the top and bottom edges
    float fTopHeight    = fTopLeft    + ((fTopRight - fTopLeft) * fPercentX );
    float fBottomHeight = fBottomLeft + ((fBottomRight - fBottomLeft) * fPercentX );

    // Calculate the resulting height interpolated between the two heights
    return fTopHeight + ((fBottomHeight - fTopHeight) * fPercentZ );
}
```

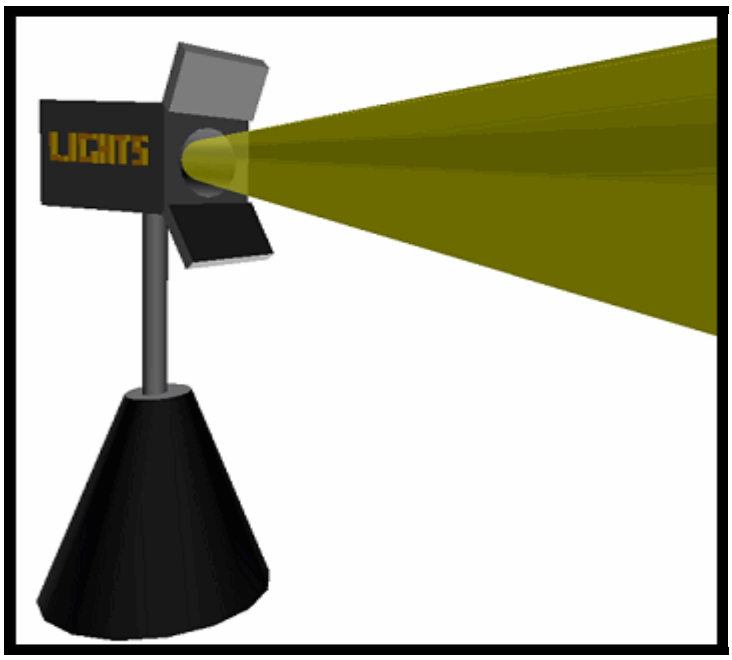
You should find the GetHeight function to be very useful in the future. Knowing how to get the height of an arbitrary position in a quad and finding height values in conjunction with height maps are important ideas. Height maps are used very often in computer games so these techniques will serve you well in later projects.

Exercises

1. What is vector regeneration and why is it necessary?
2. If you resize your viewport such that it changes the aspect ratio, do you need to rebuild the projection matrix?
3. This final exercise will demonstrate whether you have a thorough understanding of this new material. We would like you to add a third-person spacecraft mode to the camera system. This will allow the CPlayer to behave like a space craft but will have a camera that is a third person camera following the CPlayer. This might sound like you will need to create a new camera class, but actually you can use the third person camera and just make some minor adjustments to the CPlayer class. In Chapter 5 we will make available a new version of the CPlayer class that implements a third person space craft mode so that you can see if you implemented correctly. Here are a few hints that will help you on your way:

    - You will need to add a new mode to the CPlayer CAMERA_MODE enumerated type. You could call this MODE_THIRDPERSON_SC (sc= space craft).
    - You will not need to create a new CCamera derived class. You can use the CCamthirdPerson class to attach to your CPlayer when it is in this new mode.
    - You will need to make changes to the CPlayer::SetCameraMode function such that it deals with this additional mode. If the mode is MODE_THIRDPERSON_SC you will want to attach CCamthirdPerson camera to the CPlayer.
    - If you are changing modes from this new mode you will need to flatten out the CPlayer vectors just as we do now with the standard 3rd person mode.
    - You will need to add an additional option to the application camera mode menu to allow the user to switch to this new third person space craft mode.

Workbook Chapter Five: Lighting

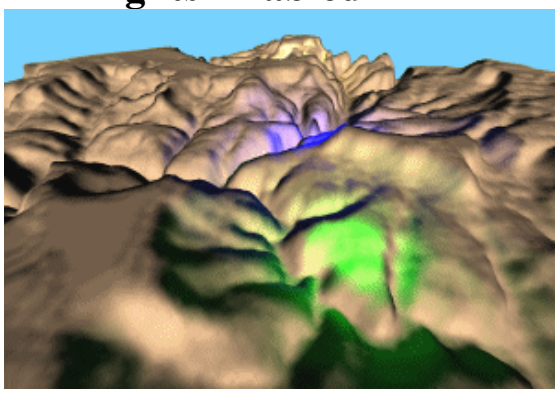


Lab Project 5.1 – Dynamic Lighting

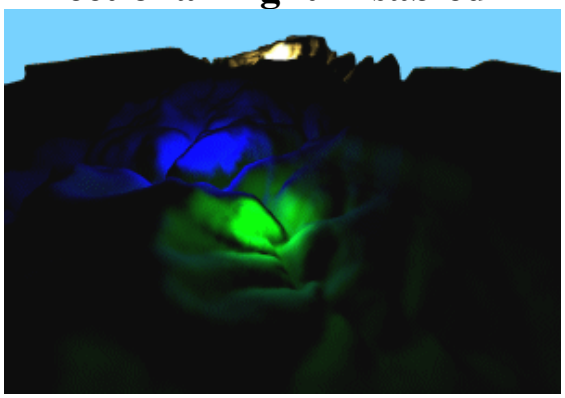
In our previous terrain demos we performed our own static vertex lighting. We generated temporary vertex normals and created a light direction vector describing the orientation of the light source. We used the dot product between these two unit length vectors to scale the light brown default color of each vertex to generate shading. Once we had the color, we stored it in the vertex and discarded the vertex normals and the light vector as they were no longer needed. Essentially, we created our own static directional light. Because we have much of the framework already in place, converting our application to use DirectX lighting is trivial. For example, we already generate the vertex normals, so all we have to do is store them in the vertices instead of discarding them. We will have to change the terrain vertex format so that it no longer stores a diffuse color but instead stores a vertex normal. Then all we have to do is add some lights to the scene, setup a default material that all vertices will use, and render the terrain using that material. As the changes to the terrain demo are very small, we will only discuss the code that has changed from Lab Project 3.2.

Lab Project 5.1 will use 5 lights to light up the terrain. The application will create one white directional light which will light the whole terrain, much like our color calculations did in Lab Project 3.2. It will also create four point lights with different colors. They will have their positions animated so that the colored lighting effects will move about the terrain. The application will also allow the user to enable or disable each light individually. The directional light is a global lighting source (much like the lighting the sun would produce in the real world), while the point lights provide localized illumination only to regions of terrain in range of their positions.

All Lights Enabled



Directional Light Disabled



Let us start digging into the code for Lab Project 5.1 beginning with the modified CVertex class (CObject.h):

```
class CVertex
{
 public:

 // Constructors
 CVertex(float fX, float fY, float fZ, const D3DXVECTOR3& vecNormal)
            { x = fX; y = fY; z = fZ; Normal = vecNormal; }
 CVertex()  { x = 0.0f; y = 0.0f; z = 0.0f; Normal = D3DXVECTOR3( 0, 0, 0 ); }

 // Public Member Functions
 float       x;          // Vertex Position X Component
 float       y;          // Vertex Position Y Component
 float       z;          // Vertex Position Z Component
 D3DXVECTOR3 Normal;     // Vertex Normal
};
```

CGameApp now includes two new members to hold the properties of the five lights in our scene:

```
D3DLIGHT9      m_Light[5];         // Lights we are using in the scene
bool           m_LightEnabled[5];  // Are lights enabled / disabled ?
D3DMATERIAL9   m_BaseMaterial;     // Base material
```

We store the light and material properties for cases where the device is lost and then reset. Just as we need to reset all render states, we also need to reset and re-enable all of the lights and materials. Notice that this application uses a single material for all faces. We will look at the properties of this material in a moment.

CGameApp::SetupGameState

The first function that has changed is the CGameApp::SetupGameState function. This function is called at application startup to setup the global state of the application. The first part of the function is unchanged; it sets up the camera and player classes, places the camera in first person mode and sets up the player physics:

```
void CGameApp::SetupGameState()
{
    // Generate an identity matrix
    D3DXMatrixIdentity( &m_mtxIdentity );

    // App is active
    m_bActive = true;

    m_Player.SetCameraMode( CCamera::MODE_FPS );
    m_pCamera = m_Player.GetCamera();

    // Setup our player's default details
    m_Player.SetFriction( 250.0f ); // Per Second
    m_Player.SetGravity( D3DXVECTOR3( 0, -400.0f, 0 ) );
    m_Player.SetMaxVelocityXZ( 125.0f );
    m_Player.SetMaxVelocityY ( 400.0f );
    m_Player.SetCamOffset( D3DXVECTOR3( 0.0f, 10.0f, 0.0f ) );
    m_Player.SetCamLag( 0.0f );
```

```
    // Set up the players collision volume info
    VOLUME_INFO Volume;
    Volume.Min = D3DXVECTOR3( -3, -10, -3 );
    Volume.Max = D3DXVECTOR3(  3,  10,  3 );
    m_Player.SetVolumeInfo( Volume );

    // Setup our cameras view details
    m_pCamera->SetFOV( 160.0f );
    m_pCamera->SetViewport(m_nViewX,m_nViewY,m_nViewWidth,m_nViewHeight,1.01f,5000.0f);

    // Set the camera volume info (matches player volume)
    m_pCamera->SetVolumeInfo( Volume );

    // Add the update callbacks required
    m_Player.AddPlayerCallback( CTerrain::UpdatePlayer, (LPVOID)&m_Terrain );
    m_Player.AddCameraCallback( CTerrain::UpdateCamera, (LPVOID)&m_Terrain );

    // Lets give a small initial rotation and set initial position
    m_Player.SetPosition( D3DXVECTOR3( 430.0f, 400.0f, 330.0f ) );
```

The above code is the same as our last demo. Next we set up a default material that will be used to render all of the terrain faces with a brownish color: ARGB (1.0, 1.0, 0.8, 0.6). This is the diffuse reflectance property of the material:

```
     // Build base white material
     ZeroMemory( &m_BaseMaterial, sizeof(D3DMATERIAL9));
     m_BaseMaterial.Diffuse.a = 1.0f; m_BaseMaterial.Diffuse.r = 1.0f;
     m_BaseMaterial.Diffuse.g = 0.8f; m_BaseMaterial.Diffuse.b = 0.6f;
```

We then set the ambient material property to reflect all ambient light. None of the lights in this demo will emit their own ambient color, so this means the terrain will have only the global ambient color (set with SetRenderState) added to the color of all faces for a base level of illumination.

```
     m_BaseMaterial.Ambient.a = 1.0f; m_BaseMaterial.Ambient.r = 1.0f;
     m_BaseMaterial.Ambient.g = 1.0f; m_BaseMaterial.Ambient.b = 1.0f;
```

The material has no emissive property and does not reflect specular light since this would not be suitable for a terrain. Notice that the specular color, power, and emissive color members are set to zero by the call to ZeroMemory before setting up the structure. At this point CGameApp::m_BaseMaterial now holds the material properties used to inform the device of the reflectance properties of the terrain.

Now we can setup each of the five lights used by the application by filling out the members of CGameApp::m_Light array (an array of D3DLIGHT9 structures). First we zero out the entire array so that we do not have to explicitly set members to zero that we do not wish to use.

```
    // Setup initial light states
    ZeroMemory( &m_Light, 5 * sizeof(D3DLIGHT9) );
```

The first light we set up is a white directional light. It only emits diffuse light and is similar to the directional light vector we used to calculate our vertex colors in Lab Project 3.2. We also set the

Boolean ‘Enabled’ flag to true so that the light is enabled by default. The user of the application can enable or disable the individual lights.

```
    // Main static directional light
    m_Light[0].Type      = D3DLIGHT_DIRECTIONAL;
    m_Light[0].Direction = D3DXVECTOR3( 0.650945f, -0.390567f, 0.650945f );
    m_Light[0].Diffuse.a = 1.0f;
    m_Light[0].Diffuse.r = 1.0f;
    m_Light[0].Diffuse.g = 1.0f;
    m_Light[0].Diffuse.b = 1.0f;
    m_LightEnabled[0]    = true;
```

The next five lights are all point lights with various positions, colors, ranges and attenuation properties. They are all enabled by default.

```
    // Players following light
    m_Light[1].Type           = D3DLIGHT_POINT;
    m_Light[1].Position       = m_Player.GetPosition();
    m_Light[1].Range          = 70.0f;
    m_Light[1].Attenuation1   = 0.02f;
    m_Light[1].Attenuation2   = 0.002f;
    m_Light[1].Diffuse.a      = 1.0f;
    m_Light[1].Diffuse.r      = 1.0f;      // Red Light
    m_Light[1].Diffuse.g      = 0.0f;
    m_Light[1].Diffuse.b      = 0.0f;
    m_LightEnabled[1]         = true;

    // Dynamic floating light 1
    m_Light[2].Type       = D3DLIGHT_POINT;
    m_Light[2].Position   = D3DXVECTOR3( 500, 0, 500 );
    m_Light[2].Position.y = m_Terrain.GetHeight(m_Light[2].Position.x,
                                                m_Light[2].Position.z)+30.0f;
    m_Light[2].Range        = 500.0f;
    m_Light[2].Attenuation1 = 0.0002f;
    m_Light[2].Attenuation2 = 0.0001f;
    m_Light[2].Diffuse.a    = 1.0f;     // Green Light
    m_Light[2].Diffuse.r    = 0.0f;
    m_Light[2].Diffuse.g    = 1.0f;
    m_Light[2].Diffuse.b    = 0.0f;
    m_LightEnabled[2]       = true;

    // Dynamic floating light 2
    m_Light[3].Type       = D3DLIGHT_POINT;
    m_Light[3].Position   = D3DXVECTOR3( 1000, 0, 1000 );
    m_Light[3].Position.y = m_Terrain.GetHeight(m_Light[3].Position.x,
                                                m_Light[3].Position.z)+30.0f;
    m_Light[3].Range        = 500.0f;
    m_Light[3].Attenuation1 = 0.000002f;
    m_Light[3].Attenuation2 = 0.00002f;
    m_Light[3].Diffuse.a    = 1.0f;
    m_Light[3].Diffuse.r    = 0.0f;
    m_Light[3].Diffuse.g    = 0.0f;
    m_Light[3].Diffuse.b    = 1.0f;        // Blue light
    m_LightEnabled[3]       = true;

    // Dynamic floating light 3
    m_Light[4].Type       = D3DLIGHT_POINT;
```

```
    m_Light[4].Position   = D3DXVECTOR3( 1500, 0, 1500 );
    m_Light[4].Position.y = m_Terrain.GetHeight(m_Light[4].Position.x,
                                                m_Light[4].Position.z)+30.0f;
    m_Light[4].Range        = 500.0f;
    m_Light[4].Attenuation1 = 0.00002f;
    m_Light[4].Attenuation2 = 0.00002f;
    m_Light[4].Diffuse.a    = 1.0f;
    m_Light[4].Diffuse.r    = 1.0f;      // Red/ blue light
    m_Light[4].Diffuse.g    = 1.0f;
    m_Light[4].Diffuse.b    = 0.5f;
    m_LightEnabled[4]       = true;
}
```

Lights 2-3 will be animated in a circular pattern as we will see when we look at the CGameApp::AnimateObjects function. Light 1 is initially set to the position of the player object and will be updated whenever the position of the player is updated. This creates a light that follows the player about the terrain. Notice that we use the CTerrain::GetHeight function to position the other point lights at a position 30 units above the terrain.

CGameApp::SetupRenderStates

As usual, the SetupRenderStates function is called at application startup and when the device is lost and then reset, to set the state of the device object. This is where we will add the code that enables lighting, sets the material and sets and enables our five lights. Most of this function should be self explanatory.

```
void CGameApp::SetupRenderStates()
{
    // Validate Requirements
    if (!m_pD3DDevice || !m_pCamera ) return;

    // Setup our D3D Device initial states
    m_pD3DDevice->SetRenderState( D3DRS_ZENABLE, D3DZB_TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_DITHERENABLE,  TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_SHADEMODE, D3DSHADE_GOURAUD );
    m_pD3DDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CCW );
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_AMBIENT, 0x0D0D0D );
```

The lighting pipeline is now enabled and the global ambient lighting value initialized.

```
    // Setup option dependant states
    m_pD3DDevice->SetRenderState( D3DRS_FILLMODE, m_FillMode );
```

We inform the device that our vertices now include a vertex normal to be used for lighting:

```
    // Setup our vertex FVF code
    m_pD3DDevice->SetFVF( D3DFVF_XYZ | D3DFVF_NORMAL );

    // Update our device with our camera details (Required on reset)
```

```
    m_pCamera->UpdateRenderView( m_pD3DDevice );
    m_pCamera->UpdateRenderProj( m_pD3DDevice );
```

Next we set the terrain material. If we did not set a material, we would not see anything on the screen because the default material reflects no light whatsoever (faces would be totally black).

```
    // Set base material
    m_pD3DDevice->SetMaterial( &m_BaseMaterial );
```

Finally, we loop through the five lights in the m_Light array and set each light as a device light at the appropriate index. We enable the light if the corresponding m_LightEnabled Boolean is set to true for a given index. These values are all set to TRUE when the application is initialized, but they can be toggled by the user.

```
    // Set and enable all lights
    for ( ULONG I = 0; I < 5; I++ )
    {
        m_pD3DDevice->SetLight( I, &m_Light[I] );
        m_pD3DDevice->LightEnable( I, m_LightEnabled[I] );
    } // Next Light
}
```

The device now has all the information it needs to correctly light our vertices. That is essentially all we would need to do under normal circumstances. But in this demo we wish to animate our lights, so we place some code in the CGameApp::AnimateObjects function to update the positions of the lights each frame.

CGameApp::AnimateObjects

This function is called once per frame to give our application a chance to update the positions of any objects. We will use it to animate the light positions. The first light (Light[0]) is not animated as it is a directional light that has no position (although we could animate the direction vector if desired). The second light (Light[1]) is the light that follows the player around the terrain. We will copy the CPlayer position into the light position and resend the light properties to the device.

```
void CGameApp::AnimateObjects()
{
    static float Angle1 = 0;
    static float Angle2 = 6.28f;

    // Update Light Positions
    m_Light[1].Position = m_Player.GetPosition();
    m_pD3DDevice->SetLight( 1, &m_Light[1] );
```

The rest of the lights rotate around a point in 3D space along a radius of 250 units. We use the sin and cosine functions to move around the circumference of a 2D circle. Light[2] has its X and Z positions calculated as being at some point on a circle in the XZ plane with its center point at (X=500, Z=500) in

world space and a radius of 240 units. The Angle1 variable is incremented each frame to allow the XZ coordinate pair to specify a new position on the circumference of the circle. The new Y position of the light is simply +30 above the terrain height which can be found using the XZ coordinate pair. Finally, we send the new light properties to the device using the SetLight function. If the light slot is currently enabled, the changes will be seen immediately. If the light is disabled, the properties of the light are still updated in the device, but the light will not be used in lighting calculations until it is enabled.

```
    m_Light[2].Position.x = 500.0f + (sinf( Angle1 ) * 250);
    m_Light[2].Position.z = 500.0f + (cosf( Angle1 ) * 250);
    m_Light[2].Position.y = m_Terrain.GetHeight(m_Light[2].Position.x,
                                                m_Light[2].Position.z)+30.0f;
    m_pD3DDevice->SetLight( 2, &m_Light[2] );
```

We use the same approach for the remaining two lights. We animate their X and Z coordinates so that they step incrementally around a circle with a radius of 250 units. For Light[3], this circle has XZ position (1000,1000). Light[4] has XZ position (1500, 1500).

```
    m_Light[3].Position.x = 1000.0f + (sinf( Angle2 ) * 250);
    m_Light[3].Position.z = 1000.0f + (cosf( Angle2 ) * 250);
    m_Light[3].Position.y = m_Terrain.GetHeight(m_Light[3].Position.x,
                                                m_Light[3].Position.z)+30.0f;
    m_pD3DDevice->SetLight( 3, &m_Light[3] );

    m_Light[4].Position.x = 1500.0f + (sinf( Angle1 ) * 250);
    m_Light[4].Position.z = 1500.0f + (cosf( Angle1 ) * 250);
    m_Light[4].Position.y = m_Terrain.GetHeight(m_Light[4].Position.x,
                                                m_Light[4].Position.z)+30.0f;
    m_pD3DDevice->SetLight( 4, &m_Light[4] );
```

Two angle variables are used to introduce some variety into the animation pattern of the lights. They are incremented/decremented so that they step through 360 degrees (0.0 to 6.28 radians).

```
    // Update angle values
    Angle1 += 0.5f * m_Timer.GetTimeElapsed();
    if ( Angle1 > 6.28f ) Angle1 -= 6.28f;

    Angle2 -= 1.0f * m_Timer.GetTimeElapsed();
    if ( Angle2 < 0.0f ) Angle2 += 6.28f;
}
```

Lastly, we look at the new case that has been added to the message processing code in CGameApp::DisplayWndProc. This case is triggered by the user selecting a menu item to enable or disable one of the five lights.

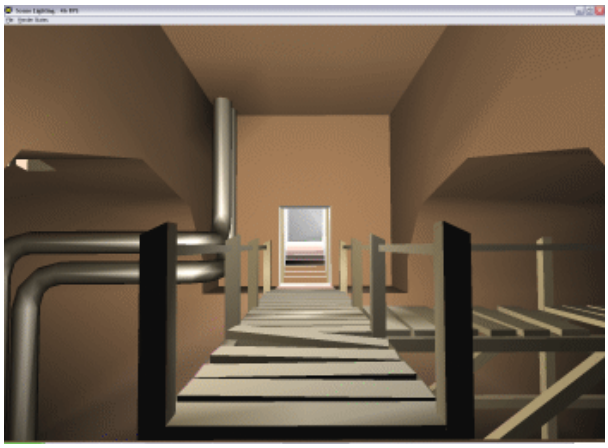
```
 case ID_LIGHT_0:
 case ID_LIGHT_1:
 case ID_LIGHT_2:
 case ID_LIGHT_3:
 case ID_LIGHT_4:

   // Enable / Disable the specified light
   ULONG LightID = LOWORD(wParam) - ID_LIGHT_0, Flags = MF_BYCOMMAND;
```

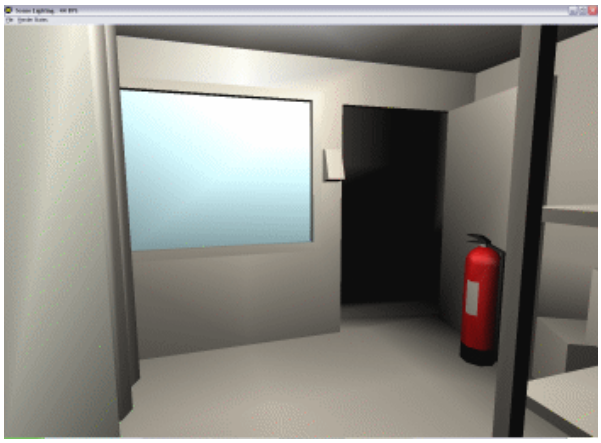
```
    m_LightEnabled[ LightID ] = !m_LightEnabled[ LightID ];
    m_pD3DDevice->LightEnable( LightID, m_LightEnabled[ LightID ] );

    // Adjust menu item
    if ( m_LightEnabled[ LightID ] ) Flags |= MF_CHECKED; else Flags |= MF_UNCHECKED;
      CheckMenuItem( m_hMenu, LOWORD(wParam), Flags );

    break;
```

Because the ID_LIGHT identifiers have sequential values and because these are the command values assigned to the lighting menu options, we can simply subtract the value of ID_LIGHT_0 from the menu ID passed in to the function in the low word of the wParam parameter. This will leave us a value between 0 and 4 describing which light needs to be enabled or disabled. Next, we toggle the selected light's Enabled state so that if it is enabled it gets disabled and vice versa. We use this state to enable/disable the light on the device. A disabled light will be ignored by the device during lighting calculations even though the settings of the light properties still remain intact (short of deliberately overwriting them with new values).

That is essentially all of the relevant code for Lab Project 5.1. Things will be somewhat more complex in the next demonstration when we implement a system that can be used to manage many lights in a scene and still abide by the maximum number of simultaneously active lights that the graphics hardware allows us to use.

Lab Project 5.2: Scene Lighting





Screenshot from Chapter5Demo2: Level created in GILES™ and stored in IWF 2.0 file format.

With a thorough discussion of the lighting pipeline behind us and the terrain demo modified to use vertex lighting, we can now attempt to load and render a fully lit 3D world. We will not bog ourselves down with file loading code in this project. If you are unfamiliar with GILES™ or the IWF file format and would like to learn how to load these levels into your game engine, please read the IWF SDK Overview document included with this lesson.

We have covered all of the basic DirectX fixed function lighting concepts. We know how to set up lights and materials and we also know how the lighting calculations work at a high level. We also learned that we can use render states to set the global ambient scene lighting or to change the color sources used by the lighting calculations.

But knowing how to set up a few lights and materials is not enough to be able to use them in a real game situation. For starters, while most game worlds include many light sources (perhaps 100s or even 1000s), the number of lights supported on a device can also vary widely across systems. Even a modern 3D card typically supports no more than 8 active lights with earlier cards supporting even fewer. If the hardware does not support lighting, then all of the lighting calculations will have to be done on the CPU by the DirectX lighting module. We will need a way to manage the many lights in a scene so that only the allowable numbers of lights are enabled at any one time. This is exactly what we will be doing in Lab Project 5.2.

We have already discussed how important it is to minimize the number of DrawPrimitive calls so that we can render as many triangles as possible in a single call (remaining within our efficiency threshold of course). This poses a problem once we start rendering triangles that use different materials. Certainly we could loop through each triangle, set the material it uses and render it with a DrawPrimitive call, but this would be a very inefficient approach. We would spend more time caught

up in function call overhead for all of the DrawPrimitive and SetMaterial calls then we would for rendering the triangles themselves.

The solution is to use batch triangles together that share the same properties. Batching is a concept that we better start getting used to, because the idea of grouping things together into chunks for faster processing is something we will do in all of our game programming projects. Doing so will allow us to minimize render state changes and maximize triangles sent to the hardware in a single DrawPrimitive call. In Lab Project 5.2, we will batch our polygons together into groups that are categorized by the material that they share. For example, we will have all the triangles that use material 1 stored together, followed by all triangles that use material 2, and so on. We minimize redundant render states by rendering all of the triangles that use a material in one go. The following list shows the rendering order using batching:

- BeginScene
- Set Material 1
- Render all triangles that use Material 1
- Set Material 2
- Render all triangles that use Material 2
- Set Material 3
- Render all triangles that use Material 3
- EndScene

Even if we had thousands of triangles in the above case, as long as we had pre-grouped them into three batches based on their material, we could render them with only three calls to SetMaterial and DrawPrimitive.

We will also implement a system that allows us to set an arbitrary number of lights in a scene. Our code will test all of the lights in the scene against the polygons in the scene as a pre-process and will create groups of polygons called “light groups”. These light groups will batch polygons together according to the groups of lights that most influence them. Since we may only be allowed to set 8 lights at one time, each light group will contain 8 lights and a list of polygons that consider those lights their most dominant color contributors. So even if we have a situation where the scene has 100 lights within range of a polygon, our code will calculate which of those lights contribute most to the resulting colors of the vertices in the polygon and use them when rendering. The number of lights in a single light group will typically be the maximum number of simultaneous lights allowed by the device. We can also set some light slots aside to use for dynamic lights.

Let us imagine that each group contains eight lights and that there are multiple light groups consisting of different light combinations. We can see that we not only have to batch by material, but also by light group as the following pseudo code shows. In this example we are assuming that the maximum number of allowable simultaneous lights is eight. We are also assuming that there are three materials used by the scene and that all polygons are affected by one of three possible light groups.

- BeginScene
- Disable any previous lights and enable all 8 lights in light group 1
- Set Material 1
- Render all triangles that use material 1 and light group 1
- Set Material 2
- Render all triangles that use material 2 and light group 1
- Set Material 3
- Render all triangles that use material 3 and light group 1
- Disable any previous lights and enable all 8 lights from light group 2
- Set Material 1
- Render all triangles that use material 1 and light group 2
- Set Material 2
- Render all triangles that use material 2 and light group 2
- Set Material 3
- Render all triangles that use material 3 and light group 2
- Disable any previous lights and enable all 8 lights from light group 2
- Set Material 1
- Render all triangles that use material 1 and light group 3
- Set Material 2
- Render all triangles that use material 2 and light group 3
- Set Material 3
- Render all triangles that use material 3 and light group 3
- End Scene

As long as we use light groups (and attach lists of polygons to those light groups) we can stay within the maximum simultaneous light limit. The level designer does not need to worry about placing lights in such a way that no more than the allowable number of active lights influences a polygon. Each polygon can simply store an index into an array of light groups describing the light group to which it belongs. The calculating of light groups will be done as a pre-process at the start of the application and we will cover this in a moment.

Material Batching

After we load in our IWF file, we will have a large array of meshes. Each mesh will have an array of surfaces and each surface will contain an array of vertices (and possibly indices). If these surfaces all have their vertices stored as triangle fans, a naïve first approach to rendering this scene might be:

```
for (every mesh in the scene)
{
        for (every surface in this mesh)
        {
                pDevice->SetMaterial   ( &Face->Material);
                pDevice->DrawPrimitive ( D3DPT_TRIANGLEFAN , Face->Vertices);
        }
}
```

While this may seem easy enough, it is not the recommended rendering strategy for the reasons we discussed earlier (too many calls to SetMaterial and DrawPrimitive).

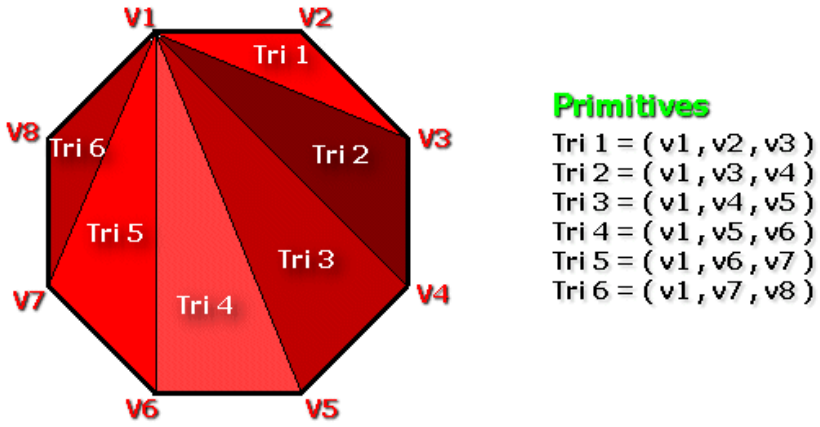
Instead we need to pre-sort our polygons into vertex buffers so that they are grouped together by the material that they use. If our scene used ten materials, we could have ten vertex buffers; one for each material. When we load our geometry, we could examine all of the faces of each mesh to see which material each one uses. This would tell us which vertex buffer this face's vertices should belong to. This means that vertex buffer[0] for example might contain all of the faces that contain material[0] in the material list (even if these faces originally existed in different meshes). Once this process is complete, we no longer have the scene stored on a per-mesh basis, but instead simply have an array of vertex buffers (one for each material). Our render loop would become:

```
BeginScene

SetMaterial 1
DrawPrimitive ( Vertex Buffer[1] )

SetMaterial 2
DrawPrimitive ( Vertex Buffer[2] )

SetMaterial 3
DrawPrimitive ( Vertex Buffer[3] )

SetMaterial 4
DrawPrimitive ( Vertex Buffer[4] )
…
etc etc
..
End Scene
```

Triangle Fans to Triangle Lists

Since we are using IWF files for this demonstration, we need to be aware of one issue. Currently, the majority of our faces are stored in CFileIWF meshes as triangle fans and as we learned in Chapter Two, triangle fans are only good for rendering connected triangles. Because a single vertex buffer in the above scheme might contain faces from many different meshes, we can no longer simply render the entire vertex buffer as a triangle fan. What we must do instead is store each face as an indexed triangle list. Now, some of the faces in the IWF file may already be stored this way, but if not, we will make the index lists ourselves. We will create an index list for each face such that every three indices describe a separate triangle which was originally part of the surface. Because these triangles are not connected to other triangles (as is the case with other primitive rendering types such as strips or fans), indexed triangles from different meshes can exist in the same vertex buffer and be rendered correctly with a single DrawIndexedPrimitive call.

The following image shows an octagon and how it may be stored as a fan-ordered list of vertices inside the IWF file (and therefore inside the iwfSurface vertex pool):



Passing these eight vertices to DrawPrimitive as a triangle fan primitive type will render six connected triangles. However, because the vertex buffer might contain many vertices from other faces -- possibly even from other meshes -- we will need to store this surface with a list of indices describing each of its triangles. We want to be able to render the entire vertex buffer as an indexed triangle list such that every three indices describes a triangle, and each triangle is completely unconnected from other triangles in the vertex buffer.

For example, imagine we have two octagon faces which belong to different meshes, but which use the same material. In this case both octagons can be placed in the same vertex buffer. The idea is not to think of these in terms of octagons anymore, but to think of them in terms of the six triangles that make up each octagon. So we would wish to build a vertex buffer that consists of the 16 vertices (8 vertices from each octagon) and an index buffer which describes 12 triangles (six for each octagon).

Av = Vertex from Polygon A (8 vertices)
Bv = Vertex from Polygon B (8 vertices)

If we created a vertex buffer big enough to hold all sixteen vertices, and if we were to copy in the first 8 vertices from polygon A and the 8 vertices from polygon B, the vertex buffer containing both of these polygons vertices would look as follows:

Pos. in Buffer 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
VertexBuff = Av1, Av2, Av3, Av4, Av5, Av6, Av7, Av8, Bv1, Bv2, Bv3, Bv4, Bv5, Bv6, Bv7, Bv8

Vertex positions 0-7 contain polygon A's vertices and vertex positions 8–15 in the vertex buffer would contain polygon B's vertices. If we were to generate indices for the octagon faces based on the diagram shown above, the triangles would be as follows:

Polygon A Index List = 0,1,2 , 0,2,3 , 0,3,4 , 0,4,5 , 0,5,6 , 0,6,7

Study the diagram to make sure that you understand this concept. Remember that index [0] references vertex v1, index2 [v1], etc. This face local index list can be copied into the global index buffer used to reference the global vertex buffer because polygon A's vertices are the first set of vertices in the vertex buffer. Although polygon B would have the same indices generated (because it is identical) it can no longer be copied into the index buffer unaltered. If that were the case, it would reference vertices 0 through 7 in the vertex buffer, which are polygon A's vertices. When we add the second polygon to the vertex buffer, we will need to add the number of vertices already stored in the vertex buffer (polygon A's vertices) to each index of the next polygon's indices. For example:

First we add polygon A's vertices to the global vertex buffer (vertices 1 through 7) and then calculate the indices for this polygon:

Polygon A Index List = 0,1,2 , 0,2,3 , 0,3,4 , 0,4,5 , 0,5,6 , 0,6,7

Now we add polygon B's vertices to the vertex buffer but we remember the vertex count that was in the vertex buffer prior to polygon B's vertices being added. We will call this variable m_OldVertexCount. Before adding polygon B's vertices to the vertex buffer, there were 8 vertices in the vertex buffer. Therefore, we first calculate the indices for polygon B local to the face which gives us the face local index list shown above. Because Polygon B's first vertex now begins at position m_OldVertexCount in the vertex buffer, we add this amount to each index in the second polygon:

Polygon B Index List = 8,9,10 , 8,10,11 , 8,11,12 , 8,12,13 , 8,13,14 , 8,14,15

This means we will have a vertex buffer with 16 vertices in it describing both polygons and a global index list describing all the triangles in that vertex buffer. This index buffer will contain 36 indices. The first 18 indices reference vertices 0 through 7. The next 18 indices reference vertices 8 through 15.

The following pseudo code shows how we could build a vertex buffer and an index buffer for every material used by the scene. If there were 10 materials used by the scene, we could have 10 vertex

buffers containing the vertices of all triangles that use that material. We could also have 10 index buffers describing how to render each of those vertex buffers as an indexed triangle list.

```
For (m = 1 to  Material Count )
{
  Create Vertex Buffer for Material m
  Create an Index Buffer for Material m

  For Each(Mesh in scene)
  {
      For Each (Surface in Mesh)
      {
            If (Surface->MaterialIndex ==1 )
            {
              m_OldVertexCount = Vertices currently in vertex buffer m
              Copy vertices from Surface into vertex buffer m
              Generate local indices for this surface (if they don't already exist)
              Add m_OldVertexCount to each of the generated indices
              Copy indices into Index Buffer m
            }
       }Next surface
    }Next mesh
 }Next material
```

At this point, we no longer have the scene stored in a per-mesh arrangement and the original meshes and surfaces can be released from memory if there is no further need for them in the application. The geometry of our scene is now stored as a series of vertex buffers grouped by material. We will examine the code to accomplish all of this as well as the code to convert many other primitive types to indexed triangle lists later in the lesson.

Light Batching

We now have geometry storage and rendering strategies that appear to be efficient -- we minimize both rendering calls and render state changes. But there is more we must consider. As mentioned previously, we need to render our triangles so that we can use the most influential lights enabled for that face so that we can work within the limited number of lights that can be simultaneously active on the device.

The number of active lights supported by the device can be checked in the device's D3DCAPS9 structure and is stored in the MaxActiveLights member. This figure is usually quite low -- typically 8 or 16 for modern hardware. If we are using a software device, then there is no actual limit on the number of lights that can be active simultaneously, but software devices are typically much slower when performing lighting calculations. Even if this was not the case, we would still need to design a system that will work within the hardware limits. In the following discussions we will use a simultaneous active limit of 8 lights.

We must also contend with the fact that different lights influence different faces according to the spatial relationships between the lights and the scene polygons. So not only do we need to make sure

that we have no more than 8 lights active when rendering our faces, but we also need to make sure that we have the correct 8 lights that are most important to each face. If we simply used the first 8 lights in the IWF file and ignored the rest, then faces that were influenced by lights 9 through 14 (for example) would receive no light and would be rendered as black.

This is obviously getting more complicated. It seems that we want to batch our polygons based on material, but we must also make sure that each face is rendered using the most influential lights. Forgetting for now exactly how we figure out which are the most influential lights (we will cover this later) and assuming that we already have this information at our disposal, the problem then becomes how we render our geometry efficiently. Certainly we still want to render polygons in batches without having to set up the lights for each face before we render it, all the while minimizing calls to SetMaterial. So in this demonstration we will create a batching approach that we will call light groups.

> **Note:** In practice, light groups work better than they sound. If a face is lit by 10 lights, we render it with a light group containing the 8 most influential lights. The discarding of the 2 least influential lights sounds like it is a bad solution but these lights will typically have a negligible effect on the color of a polygon. You will find that in nearly all levels, the designer will usually place lights so that not more than 3 or 4 lights significantly affect a single face (often fewer). If a face receives color from too many light sources simultaneously, these colors combine to make the final color bright white.

Light Groups

In our demonstration we will define a light group as a set of lights and the faces they influence. Since we have been assuming a light limit of 8, let us say that in that case, a light group would be a collection of 8 lights, and a vertex buffer containing all of the triangles that those 8 lights affect most. If there are many lights in the scene, then there may be many unique combinations of 8 sets of lights. If we built every combination, we could have hundreds of light groups or more. However, in practice, faces within the same spatial region as other faces will often share the same 8 most influential lights. There will certainly be many combinations of light sets that are not the 8 most influential to any faces in the level. Therefore, we only have to build light groups that will be used by faces. The basic approach to constructing light groups will be as follows:

- Load in scene so we have access to all meshes
- Loop through every mesh in the scene
- Loop through every face in the mesh
- Loop through every light in the scene and find the 8 most influential lights for the current face
- Search our light group list to see if a light group with these 8 lights already exists,
    - if exists, add this face to that light group vertex buffer
    - else, we have found a new light combination
        - create a new light group with these 8 lights
        - add this face to the light group

The scene is pre-processed at application startup so that it does not consume time in our main rendering loop. Of course, this scheme does not yet take materials into account but we will get to that in a moment.

With faces stored in light groups, rendering becomes batched by light group to minimize calls to DrawPrimitive and to minimize the disabling/enabling of lights. We can now render the scene one light group at a time, only having to change lights as we move to each light group. The following basic steps comprise our rendering code:

- Loop through each light group
- Disable any lights currently being used by the device.
- Enable the 8 lights of this light group
- Render the light group vertex buffer

Keep in mind that we are using 8 lights only as an example. If the D3DCAPS9::MaxActiveLights member returns a higher or lower number than this, we will adjust the number of lights each light group can handle to suit this maximum. You can also deliberately use fewer light sources per group to increase performance with the potential (although likely very low) for sacrificing image quality.

Even without seeing the details, this system still does not work as well as we would like. Although we have batched our faces by light group, we cannot simply render all the faces belonging to a light group with a single call since they may all use different materials. An inappropriate solution would be as follows:

- Loop through each light group
- Disable any lights currently used by the device
- Enable the 8 lights of this light group
- Loop through each face in this light group
- Set the material used by this face on the device
- Render this face

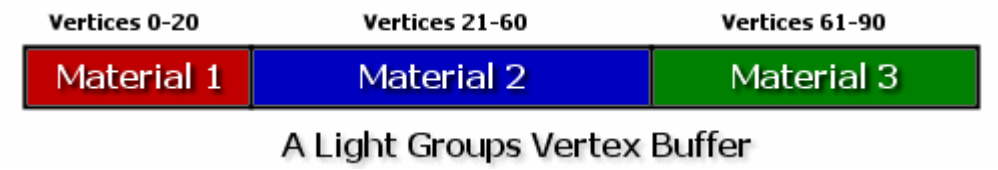
The trouble here is clear. We include no meaningful polygon batching whatsoever from a performance perspective. If there were 10,000 polygons in our scene, 10,000 SetMaterial and 10,000 DrawPrimitive calls would follow. Fortunately, the solution is rather straightforward. Instead of batching by either light group or materials, we will batch by both. We will batch first by light group and then within each of those light groups we will batch by material.

Batching Lights and Materials

We will create a light group class that contains a vertex buffer for all of the vertices of the faces that belong to a light group. When we are calculating which faces belong to which light group we will do it in the following order.

- Loop through each material
- Loop through each face
- For each face that uses this material
- Calculate the 8 most influential lights for this face and either:
    - add them to a light group which has these 8 lights, or
    - create a new light group that has these 8 lights
- Add the vertices of this face to the light group vertex buffer

As we are primarily looping and assigning faces by matching material and then adding them to the light group to which they belong, we end up with the vertices in the vertex buffer for each light group batched together by the material they use. If you step through the above routine in your head with three materials and if you imagine for simplicity that all faces belong to the same light group, you can see that all of the faces that use the first material would be placed in the light group vertex buffer first, followed by all the faces that use the second material and finally, all the vertices that use the third material. Using this method, we have one vertex buffer per light group, but when we render each group of faces, we can render them one section of the vertex buffer at a time, setting the material that the section uses before rendering it. We now have our faces batched primarily by light group, but batched in the vertex buffer by material.



Remember that the DrawPrimitive and DrawIndexedPrimitive methods of the device interface allow us to render vertex buffer in sections so this works out perfectly. Alternatively, we could just give the light group class an array of vertex buffers, one for each material it uses, but we find this approach more manageable (and it avoids the multiple calls to SetStreamSource).

We discussed earlier that for all of these different faces to exist in a single vertex buffer, we will need to render them as indexed triangles. So we will need our light group class to record the starting vertex and the vertex count of where each material's vertices start and end in the vertex buffer. This way we know which sections to render with each material. We will also need to generate index lists for each section of the vertex buffer, so in the above diagram, we would have one vertex buffer in our light group, but would need three sets of indices -- one for the triangles in each of our three sections on our vertex buffer. To help manage this system, we will create a second class which will be a child of the light group class. This class will be called a property class.

If a light group has faces in its vertex buffer that use five different materials, it will still have a vertex buffer stored at the light group level containing all the vertices as discussed above. It would also have an array of five child property classes. Each property class contains a material index, a starting position into the light group vertex buffer where the material's vertices start and a vertex count describing how

many vertices past the vertex start position use this material. It will also contain an index buffer storing the triangles in that section of the vertex buffer. Indeed we could have used a shared index buffer in the light group here as well, but we have to leave something for homework assignments right?

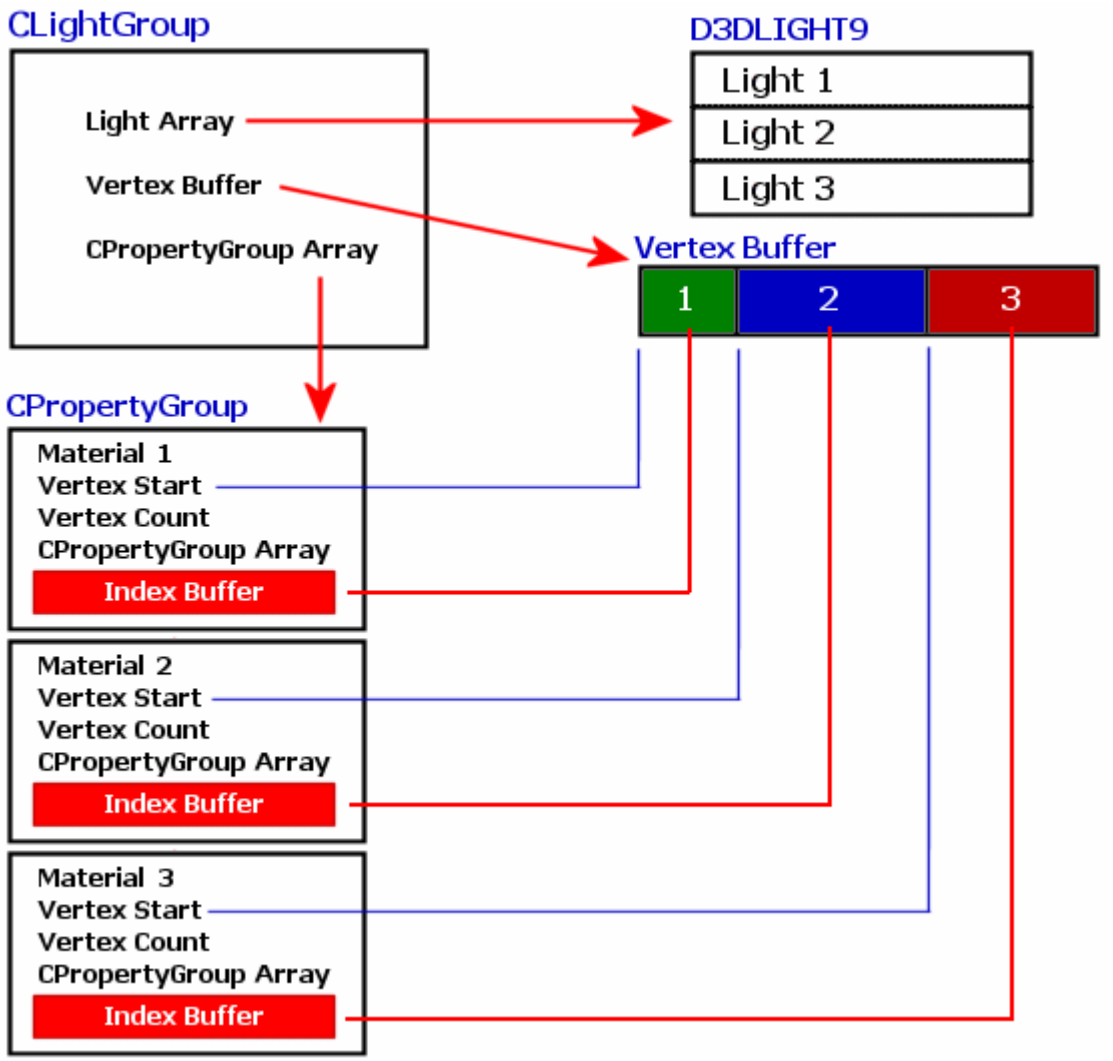
Our new rendering strategy will look like this:

- Loop through each light group
- Activate lights for this light group
- Loop through each property group of this light group
- Set the material for this property group
- Render the indexed triangle list stored at this property group

We now have a decent batching strategy beginning to come together. If we have eight light groups, then we only have to bother setting up lights eight times per frame. For each light group, we render the faces batched by material -- what we will now refer to as batching by property. The property class will be generic so that it can be used to batch by texture and/or material, and each property group can also have child property groups (although this functionality will be shown next week when we cover texturing). In this project we will be using the property class to batch only by material.

We now have some good ideas for the CLightGroup class and the CProperty class that we will need to create. We know already that the CLightGroup will manage an array of CPropertyGroup objects. The number of light groups that are created will be equal to the number of unique combinations of lights used by faces in our level. The number of lights in a light group will be equal to the maximum number of lights that the device supports (actually this is not quite true as we will discuss in a moment). Finally, the number of property groups that a light group will have in its CPropertyGroup array will be equal to the number of different materials used by the faces in the light group vertex buffer.

The following diagram describes the relationship between the CLightGroup and CPropertyGroup classes:



Each light group contains an array of lights (technically it is an array of indices into a global array of D3DLIGHT9 structures used by the entire scene). These lights are the lights we must set during our rendering loop before rendering the faces belonging to the light group. It also has a vertex buffer containing all of the vertices for all the faces belonging to this light group. The vertices are placed into the vertex buffer ordered by the material they use. Finally, the light group contains an array of CPropertyGroup objects -- one for each different material used by the faces in the vertex buffer.

Each CPropertyGroup contains a material number describing the number of the material that must be set before rendering the faces belonging to this group. It contains a VertexStart member describing the offset into the vertex buffer where the faces using this material start in the vertex buffer. There is also a VertexCount variable so we know how many vertices after the offset use this material and can be rendered with a single call to DrawIndexPrimitive. Each CPropertyGroup can contain an array of child CPropertyGroups, but this will be set to NULL in our demo since we will not be using nested property groups in this chapter. Finally, each property group will contain an index buffer describing the faces

belonging to this material using an indexed triangle list. Because we will be adding faces to the index buffer on a face-by-face basis, the values in each index buffer will be zero based for each index buffer. This might seem at first as if the indices will always describe triangles using vertices at the beginning of the vertex buffer but if you remember back to our discussion on the DrawIndexedPrimitive function, we can pass in an offset parameter to the function which is added to each index by the transformation pipeline before transforming and rendering the triangles. Below we see the parameter list to the DrawIndexedPrimitive function again to refresh your memory.

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

In all of our previous demos we have set **BaseVertexIndex** to 0 because we needed no offset added to our index buffers. But now we will use this parameter to pass in the CPropertyGroup::VertexStart value. To better understand this, imagine that we are rendering the index buffer for material 2 and that the first vertex using material 2 is at position 100 in the vertex buffer. The VertexStart member of this property group will be set to 100. Also keep in mind that the first index in the index buffer will reference vertex[0]. When we pass in 100 as **BaseVertexIndex** the pipeline will add this value to each index so that 0=100, 1=101, 2=102 and so on. So the index values in the index buffer will be correctly mapped to the correct sections in the vertex buffer.

The CLightGroup Class

The CLightGroup is defined in CScene.h with its code implementation in CScene.cpp.

```
class CLightGroup
{
public:
      // Constructors & Destructors for This Class.
      CLightGroup( );
     ~CLightGroup( );

     // Public Functions for This Class
     bool            SetLights        ( ULONG LightCount, ULONG LightList[] );
     bool            GroupMatches     ( ULONG LightCount, ULONG LightList[] ) const;
     long            AddPropertyGroup ( USHORT Count = 1 );
     long            AddVertex        ( USHORT Count = 1 );
     bool            BuildBuffers     ( LPDIRECT3DDEVICE9 pD3DDevice, bool HardwareTnL,
                                        bool ReleaseOriginals = false );
     // Public Variables for This Class
     ULONG             m_nLightCount;          // Number of lights in this group
     ULONG           * m_pLightList;           // Lights to be set active in this group
    USHORT             m_nPropertyGroupCount;     // Number of property groups stored
```

```
    USHORT               m_nVertexCount;           // Number of vertices stored
    CPropertyGroup**     m_pPropertyGroup;         // Simple array of property groups
    CVertex             *m_pVertex;                // Simple vertex array

    LPDIRECT3DVERTEXBUFFER9 m_pVertexBuffer;       // Vertex Buffer
};
```

ULONG m_nLightCount;
This holds the number of lights in the light group. Often it will be the same as the maximum active light count, but it may be less. We will discuss later how we way want to reserve a few of the device light resources so that we have one or two lights free to use for dynamic light sources. So if we had hardware that supported 8 lights simultaneously, we could for example, specify that we only want our light groups to contain 6 lights maximum. Then, we have two free light slots independent of our light group system that we can animate in our scene. This variable ultimately describes the number of light indices in the m_pLightList array.

ULONG *m_pLightList;
This is a pointer to an array of light indices. We do not actually store the D3DLIGHT9 structures in each light group but instead store all the lights used by the scene in one big D3DLIGHT9 array global to the scene. Therefore, this array holds a number of 32-bit values where each one describes the index of a light in the D3DLIGHT9 array managed by the scene.

USHORT m_nPropertyGroupCount;
This member holds the number of property groups that are in the CPropertyGroup Array. If the faces in this light group vertex buffer reference 6 different materials, this will be set to 6 -- indicating that there are 6 property groups as children of this light group.

CPropertyGroup **m_pPropertyGroup;
This is a pointer to an array of CPropertyGroup pointers. Each property group contains a material, and a group of faces that use that material.

USHORT m_nVertexCount;
This contains the number of vertices in the vertex buffer belonging to this light group.

LPDIRECT3DVERTEXBUFFER m_pVertexBuffer;
This is a pointer to the vertex buffer that contains all the vertices belonging to this light group. All the property groups will have index buffers indexing into this vertex buffer.

CVertex *m_pVertex;
When we build the light groups, the vertices will be added to this intermediate array first. Only after all of the vertices have been added to the light group will we use the CLightGroup::BuildBuffers function to copy these vertices into the vertex buffer. This is handy because we also have the choice to keep this copy of the vertices in system memory. If the device is lost and then reset, we can rebuild our vertex buffers by calling CLightGroup::BuildBuffers again. We can choose to delete this array when we build the vertex buffer but this will result in having to recalculate the light groups when the device is reset (depending on memory pool).

CLightGroup()

The constructor simply initializes all member variables to either 0 or NULL.

```
CLightGroup::CLightGroup()
{
    // Reset / Clear all required values
    m_nPropertyGroupCount     = 0;
    m_nVertexCount            = 0;
    m_nLightCount             = 0;
    m_pPropertyGroup          = NULL;
    m_pVertex                 = NULL;
    m_pLightList              = NULL;
    m_pVertexBuffer           = NULL;
}
```

~CLightGroup()

The destructor deletes each of the property groups and then deletes the property group array. We also delete the vertex array, the light list array, and release the light group vertex buffer.

```
CLightGroup::~CLightGroup()
{
    // Release our group components
    if ( m_pPropertyGroup )
    {
        // Delete all individual groups in the array.
        for ( ULONG i = 0; i < m_nPropertyGroupCount; i++ )
        {
            if ( m_pPropertyGroup[i] ) delete m_pPropertyGroup[i];
        }

        // Free up the array itself
        delete []m_pPropertyGroup;

    } // End if

    // Release flat arrays
    if ( m_pVertex ) delete []m_pVertex;
    if ( m_pLightList ) delete m_pLightList;

    // Release D3D Objects
    if ( m_pVertexBuffer ) m_pVertexBuffer->Release();

    // Reset / Clear all required values
    m_nPropertyGroupCount     = 0;
    m_nVertexCount            = 0;
    m_nLightCount             = 0;
    m_pPropertyGroup          = NULL;
    m_pVertex                 = NULL;
    m_pLightList              = NULL;
    m_pVertexBuffer           = NULL;
}
```

CLightGroup::SetLights

The SetLights function receives an array of light indices and a count describing how many indices are in the passed array and copies these into the light indices array. This function will be used to set the lights for a light group. When we find a unique combination of lights, we will need to create a new light group. After light group creation, we just call this function to pass in the numbers of the lights that belong to it. These are indices into a global D3DLIGHT9 array which we will see later.

```
bool CLightGroup::SetLights( ULONG LightCount, ULONG LightList[] )
{
    // Release previous set if any
    if ( m_pLightList ) delete m_pLightList;

    // Allocate enough room for these lights
    m_pLightList = new ULONG[LightCount];
    if (!m_pLightList) return false;

    // Store values
    m_nLightCount = LightCount;
    memcpy( m_pLightList, LightList, LightCount * sizeof(ULONG) );

    // Success
    return true;
}
```

Notice that we are careful to make sure we de-allocate the memory taken up by any previous light index list.

CLightGroup::GroupMatches

Before we create new light groups using the above function, we will first check to see if the list of lights already exists in another light group. We pass this method an array of light indices and the function will return true if the lights belonging to this group match the lights passed in the array. For each face we will find a number of lights that most influence it. This function will then see if these lights already exist in a light group. If so, the face is added to that light group. If we do not find a matching light group then a new light group will be created with this collection of lights using SetLights.

The code compares a list of light indices with the light group light index array. It returns true if a match is found. If the number of lights passed does not equal the number of lights in this light group then there is no way that this group has a matching light list and we have an early out without doing the memory compare. Next, we use the memcmp function to do a high speed compare between the two arrays. memcmp returns zero if the contents of the memory areas match.

```
bool CLightGroup::GroupMatches( ULONG LightCount, ULONG LightList[] ) const
{
    // If length does not match, neither can the list
```

```
    if ( m_nLightCount != LightCount ) return false;

    // Compare the light lists (Match even if no lights stored)
    if ( m_nLightCount > 0 ){
        if ( memcmp( m_pLightList, LightList, LightCount * sizeof(ULONG)) != 0 )
            return false;
    }

    // Matches
    return true;
}
```

CLightGroup::AddVertex

When we find a face that belongs to a light group, its vertices are eventually added to the light group vertex buffer. The vertex buffer for a light group is not built until all the vertices have been added and that is why we use a system memory vertex array to collect the vertices initially. This function is used to allocate additional space inside the CVertex array. Once we have finished adding vertices, we will copy the array into the vertex buffer. We pass in a count specifying how many elements we would like the array to grow by. This is used to allocate a new memory buffer large enough to hold the existing vertices and the new ones.

```
long CLightGroup::AddVertex( USHORT Count )
{
    CVertex * pVertexBuffer = NULL;

    // Allocate new resized array
    if (!( pVertexBuffer = new CVertex[ m_nVertexCount + Count ] )) return -1;
```

If the light group already has vertices in its vertex array, this data must be copied into the newly allocated array and the old array released:

```
    if ( m_pVertex )
    {
        // Copy old data into new buffer
        memcpy( pVertexBuffer, m_pVertex, m_nVertexCount * sizeof(CVertex) );

        // Release old buffer
        delete []m_pVertex;
    }
```

Next we add the count (how many vertices we have added) to the light group m_nVertexCount variable and point its m_pVertex pointer at the new memory buffer.

```
    // Store pointer for new buffer
    m_pVertex = pVertexBuffer;
    m_nVertexCount += Count;

    // Return first vertex
    return m_nVertexCount - Count;
}
```

Note that this function does not add the vertex data to the array. It returns the vertex count prior to the new amount being added to indicate the index of the first position in the array where the newly allocated vertices start. Our application can use this index to add the actual vertex data. The following example code assumes that we are adding three vertices stored in a temporary array (called NewVert[]) to a light group.

```
NewVertsIndex = MyLightGroup.AddVertex(3);

for ( i=0; i< 3; i++)
{
        MyLightGroup.m_pVertex[NewVertexIndex++] = NewVert[i];
}
```

CLightGroup::AddPropertyGroup

Our application will call the AddPropertyGroup function during the process of building the light groups when we wish to assign a new face to a light group which uses a material that does not match any property group already in the light group. This function works similar to the AddVertex function since it basically just resizes the property group array and returns the index of the new property group(s) that was added. The application can then use this information to access the property group at that index and set its properties.

The light group stores an array of property group pointers (not property group objects) so we first allocate a new buffer large enough to hold any existing property groups plus the new amount that we have passed in as the parameter (default = 1). We then initialize this new memory to be safe.

```
long CLightGroup::AddPropertyGroup( USHORT Count /* = 1  */ )
{
    CPropertyGroup ** pGroupBuffer = NULL;

    if (!( pGroupBuffer = new CPropertyGroup*[ m_nPropertyGroupCount + Count ] ))
         return -1;

    ZeroMemory( &pGroupBuffer[ m_nPropertyGroupCount ], Count * sizeof( CPropertyGroup* )
);
```

If property groups already exist, then we copy all of the property group pointers from the previous array into the new one and delete the previous array:

```
    if ( m_pPropertyGroup )
    {
       // Copy old data into new buffer
       memcpy(pGroupBuffer,m_pPropertyGroup,m_nPropertyGroupCount*sizeof(CPropertyGroup*));

       // Release old buffer
       delete []m_pPropertyGroup;
    }
```

After we point the m_pPropertyGroup member pointer at this new pointer array, we allocate the new property groups and add their pointers to this array:

```
    m_pPropertyGroup = pGroupBuffer;

    // Allocate new property groups
    for ( UINT i = 0; i < Count; i++ )
    {
        // Allocate new group
        if (!( m_pPropertyGroup[ m_nPropertyGroupCount ] = new CPropertyGroup() ))
           return -1;

        // Increase overall group count
        m_nPropertyGroupCount++;
    }
```

Finally we return the index of the first newly added property group so the calling application can retrieve its pointer and set its properties.

```
    // Return first group
    return m_nPropertyGroupCount - Count;
}
```

This is all fairly standard C/C++ programming which you are no doubt used to. We are showing it here so that you will have a better understanding of what each call is doing when it is called from the light group compiler function later.

CLightGroup::BuildBuffers

The BuildBuffers function is called after the vertex array has been filled. It is responsible for creating and filling the vertex buffer with the vertex data in the array. It also calls the CPropertyGroup::BuildBuffers function for each of its property groups. This instructs the property groups to build their index buffers.

We pass this function three parameters. The first is a pointer to the device for which the vertex buffer (and index buffers) will be built. The second is a Boolean indicating whether we want to use hardware or software vertex processing. The third Boolean indicates whether we want the function to delete the vertex array after its contents have been copied into the vertex buffer.

```
bool CLightGroup::BuildBuffers( LPDIRECT3DDEVICE9 pD3DDevice, bool HardwareTnL,
                                bool ReleaseOriginals )
{
    HRESULT     hRet    = S_OK;
    CVertex    *pVertex = NULL;
    ULONG       ulUsage = D3DUSAGE_WRITEONLY;

    // Should we use software vertex processing ?
    if ( !HardwareTnL ) ulUsage |= D3DUSAGE_SOFTWAREPROCESSING;

    // Release any previously allocated vertex / index buffers
```

```
    if ( m_pVertexBuffer ) m_pVertexBuffer->Release();
    m_pVertexBuffer = NULL;

    // Create our vertex buffer
    hRet = pD3DDevice->CreateVertexBuffer( sizeof(CVertex) * m_nVertexCount,
                                           ulUsage, VERTEX_FVF,
                                           D3DPOOL_MANAGED, &m_pVertexBuffer, NULL);
    if ( FAILED( hRet ) ) return false;
```

We release any previous vertex buffer if one exists and then create a vertex buffer which is large enough to hold the number of vertices in the light group vertex array. This value is stored in the member variable m_nVertexCount. The VERTEX_FVF parameter contains the flexible vertex format flags describing the vertex structure we will be using in our demonstration and is defined in CObject.h:

```
#define VERTEX_FVF      D3DFVF_XYZ | D3DFVF_NORMAL
```

If vertex buffer creation was successful, we lock the buffer to obtain a pointer to its data area so we can start copying data into the buffer.

```
    hRet = m_pVertexBuffer->Lock(0, sizeof(CVertex)*m_nVertexCount, (void**)&pVertex, 0);
    if ( FAILED( hRet ) ) return false;
```

Because we already have the vertex data stored in the vertex array in the correct format we can simply memcpy the vertex data from the array into the vertex buffer.

```
    memcpy( pVertex, m_pVertex, sizeof(CVertex) * m_nVertexCount );
```

We are done copying vertices at this point so we need to unlock the vertex buffer. We release the vertex array as well if the caller requested it.

```
    // We are finished with the vertex buffer
    m_pVertexBuffer->Unlock();

    // Release old data if requested
    if ( ReleaseOriginals )
    {
        if ( m_pVertex ) delete []m_pVertex;
        m_pVertex       = NULL;
    }
```

All that is left to do is to loop through each entry in the property group array and call its BuildBuffers function to build the index arrays for the property groups.

```
    // Build buffers for each child property group
    for ( USHORT i = 0; i < m_nPropertyGroupCount; i++ ){
          if(!m_pPropertyGroup[i]->BuildBuffers(pD3DDevice, HardwareTnL, ReleaseOriginals))
              return false;
    }

    return true;
}
```

The CPropertyGroup Class

Although we are using property groups in this chapter to store settings on a per material basis, the CPropertyGroup class can be used to group objects together using any common property. We currently have an enumerated type which is part of the CPropertyGroup namespace called **PROPERTY_TYPE** which can be used to describe exactly what is being batched on. In our next demo all of our property groups will have an m_PropertyType member with the value set to **PROPERTY_MATERIAL** because we will be using property groups to batch faces by their material property.

```
class CPropertyGroup
{
public:

     // Enumerators for this class
     enum PROPERTY_TYPE { PROPERTY_NONE = 0, PROPERTY_MATERIAL = 1, PROPERTY_TEXTURE = 2 };

    //Constructors & Destructors for This Class.
    CPropertyGroup();
    virtual ~CPropertyGroup();

    // Public Functions for This Class
    long         AddPropertyGroup ( USHORT Count = 1 );
    long         AddIndex         ( USHORT Count = 1 );
    bool         BuildBuffers     ( LPDIRECT3DDEVICE9 pD3DDevice, bool HardwareTnL,
                                    bool ReleaseOriginals = false );
    // Public Variables for This Class
    PROPERTY_TYPE    m_PropertyType;         // Type of property this is
    ULONG            m_nPropertyData;        // 32 bit property data value
    USHORT           m_nIndexCount;          // Number of indices stored
    USHORT           m_nPropertyGroupCount;  // Number of child properties
    USHORT          *m_pIndex;               // Simple index array
    CPropertyGroup **m_pPropertyGroup;       // Array of child properties
    USHORT          m_nVertexStart;     // First vertex used in the mesh vertex array
    USHORT          m_nVertexCount;     // Number of vertices used in the mesh vertex array

    LPDIRECT3DINDEXBUFFER9  m_pIndexBuffer;  // Direct3D Index Buffer
};
```

PROPERTY_TYPE m_PropertyType;
This member describes the property that the group represents. When a property group is first created this will be set to **PROPERTY_NONE** in the constructor -- meaning it has no useful information yet. In this project we will use property groups to represent faces batched by material so as soon as we create a new property group, we will be setting this value to **PROPERTY_MATERIAL**. In the next lesson we will learn to use textures and we will also want to batch faces together with regard to the texture that they use. In those cases this member will be set to **PROPERTY_TEXTURE**. We will also see property groups that have child property groups so that we can batch by multiple keys. We could for example have a light group containing five **PROPERTY_TEXTURE** property groups that batch faces that share the same texture. Each property group could then contain three child **PROPERTY_MATERIAL** groups to sort the faces again by common material within the texture property group.

ULONG m_nPropertyData;
This is a 32-bit value used to store the data that this group represents. Since our sorting criteria can be practically anything, we will use this value in conjunction with m_PropertyType to fully represent our batching. In our current application this will contain the index of the material assigned to this property group. But it could also be used to store a pointer to a texture or some other data structure. If our application checks the m_nPropertyType member variable and discovers that it is set to `PROPERTY_MATERIAL`, then it knows that the m_nPropertyData member contains an index into the application's global material list and that all the faces in this group share that common attribute.

USHORT m_nPropertyGroupCount;
The number of child property groups that this property group has in its m_pPropertyGroup array. This will be set to zero in this project because we are batching only by material. If a property group has child property groups then it is usually these children that contain the indices and the parent property group acts like a node in a hierarchical tree structure. In this case the parent will have an empty index buffer because the indices are sorted into the child property groups.

CPropertyGroup **m_pPropertyGroup;
If the m_nPropertyGroupCount member is not zero then this member will point to an array of child property group pointers.

USHORT m_nIndexCount;
This member holds the number of indices stored in this property group. In our application, each property group will hold one or more faces stored as indexed triangle lists. Each triangle will be described by three unique indices so this count should always be a multiple of three.

USHORT *m_pIndex;
As we assign the vertices from a face to a light group and copy the indices (either loaded from file or generated) of that face into the relevant property group, the indices are copied into this array. Once all indices have been added to all property groups and all light groups are complete, our application calls CLightGroup::BuildBuffers for each light group. This will spawn calls to CPropertyGroup::BuildBuffers for each of its property groups. It will take the indices stored in this array and create a new index buffer to which the indices will be copied and used for rendering. This array can be optionally released from memory at this point but it may be beneficial to maintain it so that the index buffers can be rebuilt quickly in the event of a lost device.

USHORT m_nVertexStart;
This member stores the offset into the parent light group vertex buffer where the vertices that are used by this index list begin. Property group indices always start from 0, so we pass this value into the DrawIndexedPrimitive function so that the pipeline can add it to each of our indices before accessing the vertex buffer.

USHORT m_nVertexCount;
This is the number of vertices in the vertex buffer used by this property group starting from m_nVertexStart. Together with m_nVertexStart, these two members describe the section of the light group vertex buffer that is mapped to this property group index buffer.

LPDIRECT3DINDEXBUFFER9 m_pIndexBuffer;
This is a pointer to the property group index buffer interface. The index buffer is not created until all indices have been added to the m_pIndex array. At that point they will be copied over when the application calls the CLightGroup::BuildBuffers function which in turn calls CPropertyGroup::BuildBuffers for each of its child property groups.

This class has three member functions, AddIndex, AddPropertyGroup and BuildBuffers which are identical in form and function to the AddVertex, AddPropertyGroup and BuildBuffers methods of the CLightGroup class. As such, that code will not be covered here. Please check the accompanying source code for more information.

The CGameApp Class

There are minimal changes to CGameApp in this demonstration; one new member variable which is a pointer to a CScene object. We will discuss this class momentarily.

```
CScene   m_Scene;            // Scene management class.
```

The CScene object will load our scene from an IWF file and maintain lists of geometry, materials, and lights. These are the material and light arrays that are referenced from the light and property groups. The CGameApp class will call only two of its functions, one to load the IWF file, and another to instruct the scene to render itself each frame. This demo still uses the CCamera and CPlayer classes to move about the world.

CGameApp::SetupRenderStates

The SetupRenderStates function -- called by our framework to initialize device settings on application startup and when the device is reset -- is only slightly different. We now enable specular highlights in the lighting pipeline and set a dark gray global ambient color.

```
void CGameApp::SetupRenderStates()
{
    // Validate Requirements
    if (!m_pD3DDevice || !m_pCamera ) return;

    // Setup our D3D Device initial states
    m_pD3DDevice->SetRenderState( D3DRS_ZENABLE, D3DZB_TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_DITHERENABLE,  TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_SHADEMODE, D3DSHADE_GOURAUD );
    m_pD3DDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CCW );
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_SPECULARENABLE, TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_AMBIENT, 0x0D0D0D );

    // Setup option dependant states
    m_pD3DDevice->SetRenderState( D3DRS_FILLMODE, m_FillMode );
```

```
    // Setup our vertex FVF code
    m_pD3DDevice->SetFVF( VERTEX_FVF );

    // Update our device with our camera details (Required on reset)
    m_pCamera->UpdateRenderView( m_pD3DDevice );
    m_pCamera->UpdateRenderProj( m_pD3DDevice );
}
```

CGameApp::BuildObjects

The BuildObjects function -- called by our framework to build or prepare geometry -- has been simplified. We now use CScene::Load to load an IWF file and extract the information into light groups.

```
bool CGameApp::BuildObjects()
{
    CD3DSettings::Settings * pSettings = m_D3DSettings.GetSettings();
    bool                     HardwareTnL = true;
    D3DCAPS9                 Caps;

    // Should we use hardware TnL ?
    if ( pSettings->VertexProcessingType == SOFTWARE_VP ) HardwareTnL = false;

    // Release previously built objects
    ReleaseObjects();

    // Retrieve device capabilities
    m_pD3D->GetDeviceCaps( pSettings->AdapterOrdinal, pSettings->DeviceType, &Caps );

    // Set up scenes rendering / initialization device
    m_Scene.SetD3DDevice( m_pD3DDevice, HardwareTnL );
    ULONG LightLimit = Caps.MaxActiveLights;
    if ( !HardwareTnL ) LightLimit = 0;

    // Load our scene data
    if (!m_Scene.LoadScene( _T("Data\\Colony5.iwf"), LightLimit, 1 )) return false;

    return true;
}
```

First we retrieve the settings of the device so we know whether we want software or hardware vertex processing. We will pass this information into the CScene::SetD3DDevice function so that it knows to create the vertex buffers and index buffers for the light and property groups. The function copies the passed device pointer and the HardwareTnL Boolean into CScene member variables so that they can be accessed from the rest of the CScene code.

Next, we retrieve the capabilities of the device and record the maximum simultaneous active light count that the device is capable of. This information will be passed to the CScene::Load function so that it knows to create lights groups with no more than this number of lights per group. Notice that we also pass in a third parameter to CScene::Load which tells the scene how many of the device light slots we would like to reserve for our application to use as dynamic lights. Because the light groups are

calculated as a pre-process (and this is quite a lengthy process) it means that these lights must remain static. If we wanted to move a light belonging to a light group, we would have to calculate all of the light groups again because the relationships between the lights and the polygons in the scene would have changed. This is far too slow to do during the rendering loop, so once we have calculated our light groups they remain fixed. If we want to use a dynamic light source then we can tell the light group system not to use all of the light slots available for each light group and keep some slots open for this scenario. In our application we reserve 1 light for dynamic use.

CGameApp::FrameAdvance

The rendering of the scene has been moved into CScene::Render, so the core section of the FrameAdvance function now looks like this:

```
    // Clear the frame & depth buffer ready for drawing
    m_pD3DDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, 0x79D3FF, 1.0f, 0 );

    // Begin Scene Rendering
    m_pD3DDevice->BeginScene();

    // Render the scene
    m_Scene.Render( );

    // End Scene Rendering
    m_pD3DDevice->EndScene();

    // Present the buffer
    if( FAILED(m_pD3DDevice->Present( NULL, NULL, NULL, NULL )) ) m_bLostDevice = true;
```

The CScene Class

The CScene class manages the complete scene including all geometry, materials, and lights loaded from our IWF file. It has only four public member functions which we will examine as we move along. The CScene class definition is contained in the file CScene.h.

```
class CScene
{
public:
     // Constructors & Destructors for This Class.
     CScene( );
    ~CScene( );

    // Public Functions for This Class
    void            SetD3DDevice( LPDIRECT3DDEVICE9 pD3DDevice, bool HardwareTnL );
    bool            LoadScene   ( TCHAR * strFileName, ULONG LightLimit = 0,
                                  ULONG LightReservedCount = 0 );
    void            Release     ( );
    void            Render      ( );

    // Public Variables for This Class
```

```
    D3DMATERIAL9   *m_pMaterialList;      // Array of material structures
    D3DLIGHT9      *m_pLightList;         // Array of light structures
    D3DLIGHT9       m_DynamicLight;       // Single dynamic light for testing
    CLightGroup   **m_ppLightGroupList;   // Array of individual lighting groups
    ULONG           m_nMaterialCount;     // Number of materials stored
    ULONG           m_nLightCount;        // Number lights stored here
    ULONG           m_nLightGroupCount;   // Number of light groups stored here

private:
   // Private Functions for This Class
    bool ProcessMeshes(CFileIWF & pFile);
    bool ProcessVertices(CLightGroup *pLightGroup,
                         CPropertyGroup *pProperty,iwfSurface *pFilePoly);
    bool ProcessIndices(CLightGroup *pLightGroup,
                        CPropertyGroup *pProperty,iwfSurface *pFilePoly);
    bool ProcessMaterials(const CFileIWF& File);
    bool ProcessEntities(const CFileIWF& File);
    float GetLightContribution (iwfSurface *pSurface, D3DLIGHT9 *pLight);
    long  AddLightGroup(ULONG Count);
    bool  BuildLightGroups(std::vector<iwfSurface*> &SurfaceList, long MaterialIndex);

    // Private Variables for This Class
    ULONG               m_nReservedLights;  // Number of light slots to leave empty
    ULONG               m_nLightLimit;   // Number of device lights available
    LPDIRECT3DDEVICE9   m_pD3DDevice;    // D3D device used for rendering / initialization
    bool                m_bHardwareTnL;  // Objects should be build taking into account TnL
};
```

D3DMATERIAL9 *m_pMaterialList;
This member will point to an array of all materials used by the scene. The LoadScene function will call the private ProcessMaterials function to extract all of the material data from the IWF file and store it in this array. Each property group will contain a material index into this scene array of materials.

D3DLIGHT9 *m_pLightList;
This member will point to an array of all lights contained in the IWF file used by the scene. Each light group will contain an array of light indices that index into this array.

D3DLIGHT9 m_DynamicLight;
In our application we will reserve one light to be used as a dynamic light and this member holds the D3DLIGHT9 information and settings for this light. This light will always be in device slot zero and our light groups will use light slots 1 – MaxActiveLights. This light will be updated each frame as our application updates its position and resends it to the device. If you wanted to use more than one dynamic light you would want to make this an array.

CLightGroup **m_ppLightGroupList;
This member is an array of all scene CLightGroup pointers. The array contains all of the geometry in the scene divided among light groups, and further divided into property groups based on material.

ULONG m_nMaterialCount;
The number of materials in the material array -- equal to the number of materials stored in the IWF file.

ULONG m_nLightCount;
The number of lights in the light array -- equal to the number of lights entities stored in the IWF file.

ULONG m_nLightGroupCount;
The number of light groups that were built to represent the scene during the light group building process.

ULONG m_nReservedLights;
This member will contain the number of reserved lights that will be used by the application. This is used to limit the maximum number of lights stored in our light groups to **`MaxActiveLights - m_ReservedLights.`**

ULONG m_nLightLimit;
The value stored in this member will be the MaxActiveLights value describing how many active lights the device supports at any one time. This is used along with the m_nReservedLights member to calculate the maximum number of lights that can be stored in a light group.

LPDIRECT3DDEVICE9 m_pD3DDevice;
This is a pointer to the rendering device.

bool m_bHardwareTnL;
This Boolean is set to true or false depending on whether we are using hardware or software vertex processing. The scene needs this information to correctly build the vertex and index buffers so that DirectX can place them in the appropriate memory pool.

CScene::CScene()

The constructor initializes all values to zero or null and also sets up the parameters for the one dynamic light that our scene will use. We create a red light source and place it at position (320, 10, 500) -- the main hanger area of the level. These settings are stored in the m_DynamicLight member variable.

```
CScene::CScene()
{
    // Reset / Clear all required values
    m_nLightLimit      = 0;
    m_nReservedLights  = 0;

    m_nMaterialCount   = 0;
    m_nLightCount      = 0;
    m_nLightGroupCount = 0;
    m_pMaterialList    = NULL;
    m_pLightList       = NULL;
    m_ppLightGroupList = NULL;
    m_pD3DDevice       = NULL;
    m_bHardwareTnL     = false;

    // Set up our dynamic light properties
    ZeroMemory( &m_DynamicLight, sizeof(D3DLIGHT9) );
    m_DynamicLight.Type      = D3DLIGHT_POINT;
```

```
    m_DynamicLight.Range     = 150.0f;
    m_DynamicLight.Diffuse.a = 1.0f;
    m_DynamicLight.Diffuse.r = 1.0f;
    m_DynamicLight.Position  = D3DXVECTOR3( 290, 10, 500 );
    m_DynamicLight.Attenuation0=1.0;
}
```

CScene::~CScene()

The destructor calls the CScene::Release function to release all of the arrays allocated to hold the materials, lights, and light groups. This means the arrays can be destroyed either when the object is destroyed or if the application explicitly calls the CScene::Release member function.

```
CScene::~CScene()
{
    // Release allocated resources
    Release();
}
```

CScene::Release

The first thing this function does is release the light groups array. Since it is an array of pointers we loop through each element in the array and delete it first, and then we delete the actual pointer array as shown below.

```
void CScene::Release( )
{
    ULONG i;

     // Release any allocated memory
     if ( m_ppLightGroupList )
     {
         for ( i = 0; i < m_nLightGroupCount; i++ )
         {
             if ( m_ppLightGroupList[i] ) delete m_ppLightGroupList[i];
         }
         delete []m_ppLightGroupList;
     }
```

Next we delete the material and light arrays and call release on the device to decrease the reference count.

```
    // Release the materials array
    if ( m_pMaterialList ) delete []m_pMaterialList;

    // Release the lights array
    if ( m_pLightList )     delete []m_pLightList;
```

```
    // Release Direct3D Objects
    if ( m_pD3DDevice ) m_pD3DDevice->Release();
```

Finally, we set all members to zero or null.

```
    // Clear Variables
    m_nMaterialCount      = 0;
    m_nLightCount         = 0;
    m_nLightGroupCount    = 0;
    m_pMaterialList       = NULL;
    m_pLightList          = NULL;
    m_ppLightGroupList    = NULL;
    m_pD3DDevice          = NULL;
    m_bHardwareTnL        = false;
}
```

CScene::LoadScene

This function uses a CFileIWF object (one of the IWF SDK objects) to load the IWF file data:

```
bool CScene::LoadScene(TCHAR *strFileName, ULONG LightLimit /* = 0 */,
                       ULONG LightReservedCount /* = 0 */)
{
        CFileIWF File;

        // Attempt to load the file
        File.Load( strFileName );
```

Once the IWF file has been loaded into memory, its data exists in the CFileIWF internal STL vectors (see IWF Overview document included with this lesson). We extract the scene information from the CFileIWF object through the following two function calls:

```
        // Copy over the entities and materials we want from the file
        if (!ProcessEntities( File )) return false;
        if (!ProcessMaterials( File )) return false;
```

ProcessEntities is responsible for looping through the CFileIWF entity vector and copying lights into the m_pLightList array. When this function returns, all of the lights that were in the IWF file will have their information in the m_pLightList array stored in **D3DLIGHT9** format. The ProcessMaterials function works the same way. It extracts all materials from CFileIWF and copies them into the m_pMaterialList array where they are stored in **D3DMATERIAL9** format ready for use by the device.

```
        // Store values
        m_nLightLimit     = LightLimit;
        m_nReservedLights = LightReservedCount;
```

If the light limit has a value of zero, then the calling application does not want to limit the number of simultaneously active lights in any way. This can be useful if we are using a software vertex processing device which does not have a maximum simultaneous light limit. When using a software

vertex processing device, if a face is influenced by 100 lights, we can set all 100 lights (at the cost of severe performance degradation for real-time play) and enable them on the device simultaneously. When there is no light limit we simply set the light limit to the total number of lights in the scene plus the number of reserved lights. In this case a light group *could* contain every light in the scene although it would be very unlikely that a face would be affected by all of the lights in the scene.

```
        // Check for unlimited light sources
        if ( m_nLightLimit == 0 ) m_nLightLimit = m_nLightCount + LightReservedCount;
```

Next we call the ProcessMeshes member function to extract the mesh data from the CFileIWF object and build all of the light groups.

```
        // Now process the meshes and extract the required data
        if (!ProcessMeshes( File )) return false;
```

When this function returns, all light groups have been created and all faces have been assigned to the relevant light/property groups. Now we loop through each light group and call its BuildBuffers function which will copy all the vertices stored in its CVertex array into the final vertex buffer. Each light group also calls the BuildBuffers function of each of its property groups instructing them to build their index buffers.

```
        // Build vertex / index buffers
        for ( USHORT i = 0; i < m_nLightGroupCount; i++ )
        {
            if (!m_ppLightGroupList[i]->BuildBuffers(m_pD3DDevice, m_bHardwareTnL, true))
               return false;
        }
```

All data has been extracted from the CFileIWF object now so we can instruct it to free up its internal arrays.

```
        // Allow file loader to release any active objects
        File.ClearObjects();

        // Success!
        return true;
}
```

CScene::ProcessEntities

The ProcessScene function retrieves light information from the passed CFileIWF object. After we have retrieved the number of lights in the entity vector, we allocate a D3DLIGHT9 array (m_pLightList) large enough to hold them. Then we extract the information from this vector and store it in our newly allocated CScene::m_pLightList array. Note that we cannot simply retrieve the number of lights by using the STL vector ‘size’ function because the vector may contain other entity types. We are only interested in entities that have the ENTITY_LIGHT ID as shown below.

```
bool CScene::ProcessEntities( const CFileIWF& File )
{
    D3DLIGHT9 Light;
    ULONG     i;
    ULONG     LightCount = 0;

    for ( i = 0; i < File.m_vpEntityList.size(); i++ )
    {
        // Retrieve pointer to file entity
        iwfEntity * pFileEntity = File.m_vpEntityList[i];

        // Only build if this is a light entity
        if ( pFileEntity->EntityTypeID == ENTITY_LIGHT && pFileEntity->DataSize > 0 )
            LightCount++;
    }
```

We loop through each entity and increase the local LightCount variable if the entity is a light and has a data size which is not zero (this is just for safety -- we should never encounter a light entity with a 0 data size member with scenes exported from GILES). Next, we check if the light count is zero. If so, then the scene contains no lights and we can exit the procedure. Otherwise, we allocate the CScene::m_pLightList array large enough to hold *LightCount* lights.

```
    // Detect no-op
    if ( LightCount == 0 ) return true;

    // Allocate enough space for all our lights
    m_pLightList = new D3DLIGHT9[ LightCount ];
    if (!m_pLightList) return false;
```

We can now loop through each element in the CFileIWF::m_vpEntityList vector and extract the information from any light entities we find.

```
    // Loop through and build our lights
    for ( i = 0; i < File.m_vpEntityList.size(); i++ ){

        // Retrieve pointer to file entity
        iwfEntity * pFileEntity = File.m_vpEntityList[i];
```

If the entity ID indicates a light then we copy all of the parameters we need into the local D3DLIGHT9 variable *Light*.

```
        if ( pFileEntity->EntityTypeID == ENTITY_LIGHT && pFileEntity->DataSize > 0 )
        {
            LIGHTENTITY *pFileLight = (LIGHTENTITY*)pFileEntity->DataArea;
```

We use the IWF SDK LIGHTENTITY structure to access the data area of the light entity. First we check that this light is of a type our application will use. Our application will not use ambient lights because we will be setting the ambient light level using a render state as we discussed earlier.

```
            // Skip if this is not a valid light type (Not relevant to the API)
            if ( pFileLight->LightType == LIGHTTYPE_AMBIENT ) continue;
```

If we get here, then the light is a point light, a spot light, or a directional light, so we copy the information into the local D3DLIGHT9 variable. First we extract the diffuse, ambient and specular colors of the light.

```
            // Extract the light values we need
            Light.Type     = (D3DLIGHTTYPE)(pFileLight->LightType + 1);
            Light.Diffuse  = D3DXCOLOR(pFileLight->DiffuseRed, pFileLight->DiffuseGreen,
                                       pFileLight->DiffuseBlue, pFileLight->DiffuseAlpha );
            Light.Ambient  = D3DXCOLOR(pFileLight->AmbientRed, pFileLight->AmbientGreen,
                                       pFileLight->AmbientBlue, pFileLight->AmbientAlpha );
            Light.Specular = D3DXCOLOR(pFileLight->SpecularRed, pFileLight->SpecularGreen,
                                     pFileLight->SpecularBlue, pFileLight->SpecularAlpha );
```

Next, we copy the position and orientation of the light. This information is represented in a world matrix stored within the entity. We extract the position information from the 4th row of the matrix and the direction vector from the 3rd row of the world matrix.

```
            Light.Position = D3DXVECTOR3( pFileEntity->ObjectMatrix._41,
                                          pFileEntity->ObjectMatrix._42,
                                          pFileEntity->ObjectMatrix._43 );

            Light.Direction = D3DXVECTOR3( pFileEntity->ObjectMatrix._31,
                                           pFileEntity->ObjectMatrix._32,
                                           pFileEntity->ObjectMatrix._33 );
```

Finally, we extract the remaining light information such as range and attenuation which may be relevant to this light type.

```
            Light.Range         = pFileLight->Range;
            Light.Attenuation0  = pFileLight->Attenuation0;
            Light.Attenuation1  = pFileLight->Attenuation1;
            Light.Attenuation2  = pFileLight->Attenuation2;
            Light.Falloff       = pFileLight->FallOff;
            Light.Theta         = pFileLight->Theta;
            Light.Phi           = pFileLight->Phi;
```

We add the new light to our light array and increase our CScene::m_nLightCount variable so that it correctly tracks how many lights are in the array.

```
            // Add this to our vector
            m_pLightList[ m_nLightCount++ ] = Light;
        } // End if light
    } // Next Entity

    // Success!
    return true;
}
```

After the above function has been called from CScene::LoadScene we have all the lights stored in the CScene light array. Next the LoadScene function calls ProcessMaterials to extract the materials from the CFileIWF object into the CScene::m_pMaterials array.

CScene::ProcessMaterials

This function checks the size of the CFileIWF m_vpMaterialList vector, updates the CScene::m_nMaterialCount, and allocates the CScene::m_pMaterialList array to hold that many D3DMATERIAL9 structures.

```
bool CScene::ProcessMaterials( const CFileIWF& File )
{
    ULONG i;

    // Allocate enough room for all of our materials
    m_pMaterialList = new D3DMATERIAL9[ File.m_vpMaterialList.size() ];

    m_nMaterialCount = File.m_vpMaterialList.size();
```

We now loop through every material in the CFileIWF materials vector and copy the relevant information into the newly allocated CScene material list array.

```
    // Loop through and build our materials
    for ( i = 0; i < File.m_vpMaterialList.size(); i++ )
    {
        // Retrieve pointer to file material
        iwfMaterial * pFileMaterial = File.m_vpMaterialList[i];

        // Retrieve pointer to our local material
        D3DMATERIAL9 * pMaterial = &m_pMaterialList[i];

        // Copy over the data we need from the file material
        pMaterial->Diffuse  = (D3DCOLORVALUE&)pFileMaterial->Diffuse;
        pMaterial->Ambient  = (D3DCOLORVALUE&)pFileMaterial->Ambient;
        pMaterial->Emissive = (D3DCOLORVALUE&)pFileMaterial->Emissive;
        pMaterial->Specular = (D3DCOLORVALUE&)pFileMaterial->Specular;
        pMaterial->Power    = pFileMaterial->Power;

    } // Next Material

    // Success!
    return true;
}
```

CScene::ProcessMeshes

The ProcessMeshes function is responsible for assigning mesh surfaces to their appropriate light groups sorted in material order. One thing to note before examining the code is that an IWF file may contain faces with no materials applied to them. While this is not possible with surfaces exported using GILES, it is a possibility if another 3rd party IWF exporting application is used. Because of this possibility we will start our material loop at –1 instead of zero and use this initial pass through the loop to collect all surfaces that have no materials. When we then send this collection to the BuildLightGroups function they will end up in a light group that has no lights. It is important to have a

light group that has no lights so that we have somewhere to store geometry that either A) has no material applied or B) has a material applied but is not lit by any light sources. When we render polygons in this group they will appear completely black. It is quite possible that the level designer may have one or two faces not affected by any light sources in a level and wants them to appear black, so we must allow for these surfaces. Discarding them would create holes in the level where those faces used to be. In the case of the faces with no materials, you could change this code to create a default white material for such faces, but our code treats them like the unlit faces that they will share a light group with.

First we declare an STL vector to hold iwfSurface structure.

```
bool CScene::ProcessMeshes( CFileIWF & pFile )
{
    long i, j, k;
    std::vector<iwfSurface*> SurfaceList;
```

Then we loop through each material (starting at –1 since the first iteration will be used to catch surfaces with no materials).

```
    for ( i = -1; i < (signed)m_nMaterialCount; i++ )
    {
```

Next we loop through each mesh and then every face belonging to that mesh and get a pointer to the current iwfSurface we are testing.

```
        for ( j = 0; j < pFile.m_vpMeshList.size(); j++ )
        {
            iwfMesh * pMesh = pFile.m_vpMeshList[j];
            for ( k = 0; k < pMesh->SurfaceCount; k++ )
            {
                iwfSurface * pPoly = pMesh->Surfaces[k];
```

If the material loop is currently at -1 then this is the initial sweep through the outer loop where we search for faces with no materials. We do this by testing if the surface has the SCOMPONENT_MATERIALS component flag set, indicating that the surface stores a valid material index. If not, or if this surface has a channel count of zero, then the surface does not have a material assigned to it. In that case we add it to the vector and continue to test the next surface in the loop.

```
                if ( i == -1 )
                {
                    if(!(pPoly->Components & SCOMPONENT_MATERIALS) ||
                         pPoly->ChannelCount == 0)
                    {
                        SurfaceList.push_back( pPoly );
                        continue;

                    } // End if no material properties

                } // End if processing null materials
```

If we are not in the initial iteration of the materials loop (in other words i > -1) then we have a surface that does have a material. We need to check whether this surface's material index matches the material we are currently collecting surfaces for. If so, then we add the current surface to the vector.

```
                // If the material matches, add it to our list
                if ( pPoly->MaterialIndices[0] == i ) SurfaceList.push_back( pPoly );

            } // Next Surface

        } // Next Mesh
```

At this point in the material loop we have collected all of the faces from all of the meshes that use the current material into the SurfaceList vector. If there is at least one surface in this vector, we will call the BuildLightGroups function to assign them to the relevant light/property groups to which they belong.

```
        // Build our scene light groups from this sorted list.
        if ( SurfaceList.size() > 0 )
        {
            if (!BuildLightGroups( SurfaceList, i )) return false;
        }
```

At this point, the surfaces using the current material have all been assigned to light groups, so we empty the vector and it can be used in the next iteration of the material loop.

```
        // Clear our surface list
        SurfaceList.clear();

    } // Next Material
```

All light groups now have their property groups created and the surfaces have been assigned. We can return program flow back to CScene::LoadScene where it will build the vertex buffers and index buffers for each light group and hand flow back to the main application.

```
    // Success!!
    return true;
}
```

Here is the ProcessMeshes function again in its entirety so that you can read it without any interruptions.

```
bool CScene::ProcessMeshes( CFileIWF & pFile )
{
    long i, j, k;
    std::vector<iwfSurface*> SurfaceList;

    // Here we must sort our scene polygons, by material, into lists
    // We start from -1 to still sort those that have no material
    for ( i = -1; i < (signed)m_nMaterialCount; i++ )
    {
        // Now we must search for all surfaces which use this material
```

```
        for ( j = 0; j < pFile.m_vpMeshList.size(); j++ )
        {
            iwfMesh * pMesh = pFile.m_vpMeshList[j];
            for ( k = 0; k < pMesh->SurfaceCount; k++ )
            {
                iwfSurface * pPoly = pMesh->Surfaces[k];

                // If the surface has no material properties and we are
                // processing material -1, add this to that list
                if ( i == -1 )
                {
                    if ( !(pPoly->Components & SCOMPONENT_MATERIALS) ||
                           pPoly->ChannelCount == 0 )
                    {
                        SurfaceList.push_back( pPoly );  continue;
                    }
                }

                // If the material matches, add it to our list
                if ( pPoly->MaterialIndices[0] == i ) SurfaceList.push_back( pPoly );

            } // Next Surface

        } // Next Mesh

        // Build our scene light groups from this sorted list.
        if ( SurfaceList.size() > 0 )
        {
            if (!BuildLightGroups( SurfaceList, i )) return false;
        }

        // Clear our surface list
        SurfaceList.clear();

    } // Next Material

    return true;
}
```

CScene::BuildLightGroups

We will discuss the BuildLightGroups function as a three step process. Keep in mind that we have passed in the current material index that we are processing (from the ProcessMeshes function) along with a vector containing the faces that use the material. Step 1 has the job of allocating two arrays: a light contribution table that will be used to record the scores of how influential all the lights in the scene are to each face, and a selected lights table that will be used to record the most influential lights in the light contribution table for the face that we are currently processing. Once we have the selected lights for a face, we enter Step 2 which has the job of finding whether a light group already exists that includes these selected lights. If a light group is found, then the face is added to that light group and we search the light group property groups to see if a property group exists within that light group that is mapped to the current material we are processing. If a property group is found then the face is added to the property group. If not, a new property group is added to the light group which has the material

index we are currently processing along with the faces. If we cannot find a light group, then a new one will be created and the selected lights will be stored. We will add a property group to this new light group which has the material index we are currently processing. Finally the face will be added to this new light group/property group. We do this for each face in the vector passed. Step 3 copies the vertices of the face into the light group vertex buffer and copy the indices into the property group which is mapped to the current material we are processing.

Step 1: Determining Light Influence

We begin by allocating a float array called LightContribution which will be large enough to hold a single float value for every light in the scene. We will also allocate a second ULONG array large enough to hold the maximum number of lights that are allowed to exist in a single light group.

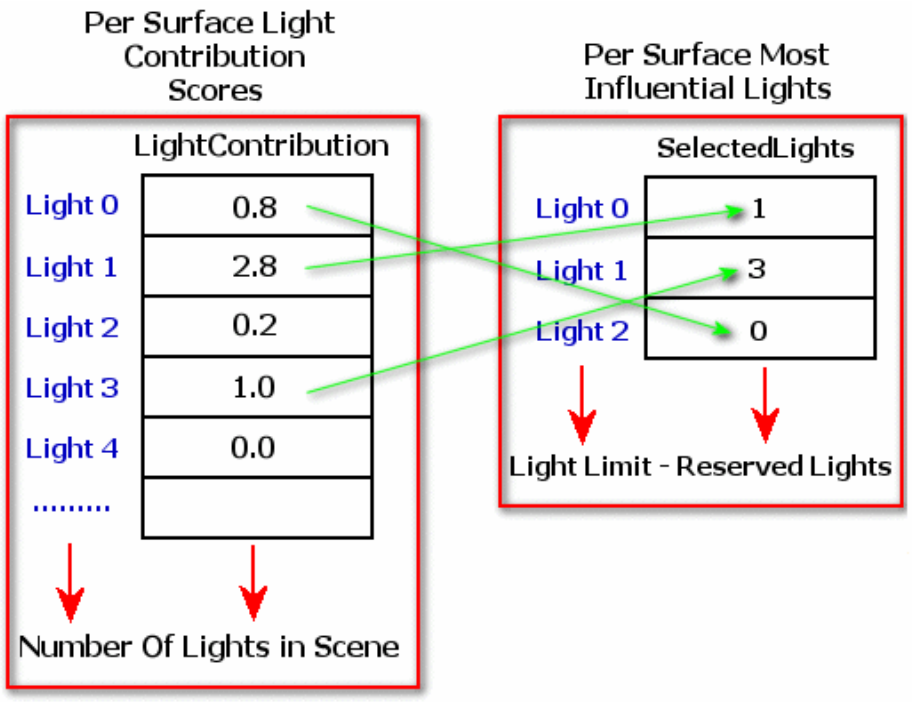
```
bool CScene::BuildLightGroups( std::vector<iwfSurface*> & SurfaceList, long MaterialIndex )
{
    ULONG           i, j, k, *SelectedLights = NULL, LightCount = 0;
    float          *LightContribution        = NULL, BestScore = 0.0f;
    CLightGroup    *pLightGroup  = NULL;
    CPropertyGroup *pProperty    = NULL;
    long            BestLight    = -1;

    // Setup our light contribution tables
    LightContribution = new float[ m_nLightCount ];
    if (!LightContribution) goto BuildFailure;

    SelectedLights = new ULONG[ (m_nLightLimit - m_nReservedLights) ];
    if (!SelectedLights) goto BuildFailure;
```

m_nLightLimit holds the maximum number of simultaneous lights allowed by the device. We must subtract the number of lights the application would like to reserve for its own uses to obtain how many lights can exist in a single light group.

The next step is to loop through every face in the passed STL vector. For each face, we loop through every light in the scene and record an influence score in the LightContribution table. At the end of the light loop, we have a score for every light describing how influential it is to the final color of the face. We call the GetLightContribution function to return an influence score for each light with the current face we are processing. This function will be covered in the next section. For now just know for now that it will typically return a value between –2.0 and +3.0 where a higher value indicates that the light influences the surface to a higher degree.

```
    // Loop through each Mesh
    for ( i = 0; i < SurfaceList.size(); i++ )
    {
        iwfSurface * pSurface = SurfaceList[i];

        // Now we will determine which lights affect this surface
        ZeroMemory( LightContribution, m_nLightCount * sizeof(float));
        for ( j = 0; j < m_nLightCount; j++ ){
            LightContribution[j] = GetLightContribution( pSurface, &m_pLightList[j] );
        }
```

At this point, if there were 100 lights in the scene, we would have 100 influence scores. Our next job is to loop through this score table and store the index of the most influential lights (those with the highest score) in our SelectedLights array. At the end of the loop the SelectedLights array will describe the lights that most influence the surface, as shown in the following diagram:



Notice that we zero out the LightContribution array for each face that we are testing, since this is a per-face process. We loop though each slot in the SelectedLights array with the intention of finding the best light to put in it. During each loop iteration, we set a local variable called *BestScore* to zero, and then loop through every light in the scene. If we find a light with a higher score than the current best score we record its score and its index. At the end of the light loop, we have the index of the best light and we copy it into the SelectedLights slot that we are currently processing. Then we set the score of this best light to 0 in the *LightContribution* array so that in the next iteration of the loop, we do not get the same best light again. Instead we get the second best light and copy that into the selected lights array. We repeat this process until we have enough lights that influence the surface to fill up the *SelectedLights* array or until we run out of lights.

```
        // Now we have the light contribution table, we can select
        // the best lights for the job (with an acceptable error)
        LightCount = 0;
        for ( j = 0; j < (m_nLightLimit - m_nReservedLights); j++ )
        {
            // Reset our best score
            BestScore = 0.0f;
            BestLight = -1;
```

```
            // Find the light with the best score
            for ( k = 0; k < m_nLightCount; k++ )
            {
                if ( LightContribution[ k ] > BestScore )
                {
                    BestScore = LightContribution[ k ];
                    BestLight = k;
                }
            } // Next Light

            // Have we run out of lights ?
            if ( BestLight < 0 ) break;

            // Select our best light. We reset its score here.
            SelectedLights[ LightCount++ ] = BestLight;
            LightContribution[ BestLight ] = 0.0f;

        } // Next Light Slot
```

At this point we have an array of lights (this can never be more than our light limit minus the reserved light count) which describes the lights that our surface should share a light group with.

Step 2: Finding a Light Group

We now search the CScene light group array to find a light group that matches the set of lights we have in the *SelectedLights* array from Step 1. We call CLightGroup::GroupMatches to compare the light indices in the *SelectedLights* array with the light indices stored within the light group. It returns true if the light indices match. If they do match, we will remember this light group (using the local *pLightGroup* pointer) so that we can use this pointer to search its property groups (more on this in a moment). The local variable *LightCount* contains the number of lights in the *SelectedLights* array.

```
        pLightGroup = NULL;
        for ( j = 0; j < m_nLightGroupCount; j++ )
        {
            if ( m_ppLightGroupList[j]->GroupMatches( LightCount, SelectedLights ) )
            {
                // Select this light group and bail
                pLightGroup = m_ppLightGroupList[j];
                break;
            } // End if group matches
        } // Next Light group
```

If we could not find a light group with the correct combination of lights in it then we have to create a new light group for this light set. We do this by first allocating a new light group and then calling CScene::AddLightGroup which resizes the CScene light group array to make space for another light group pointer at the end. We copy the new light group pointer onto the end of the array and use the SetLight function to pass in the selected lights (we covered this function earlier). This function extracts the lights out of the *SelectedLights* array and into the light group. It is important to realize that all of our light groups will be created here since this is the only place in the application where light groups are selected.

```
        // If we didn't find a light group, allocate and add one
        if ( !pLightGroup )
        {
            if (!(pLightGroup = new CLightGroup) ) goto BuildFailure;

            // Add it to the list
            if ( AddLightGroup( 1 ) < 0 ) goto BuildFailure;
            m_ppLightGroupList[ m_nLightGroupCount - 1 ] = pLightGroup;
            pLightGroup->SetLights( LightCount, SelectedLights );
        }
```

The local pointer pLightGroup now points to a pre-existing light group or to one which was newly created. Also remember that we passed in a material index that describes the material the current face is using. Now it is time to search the light group's CPropertyGroup array to try to find a property group which is already using this material. If one is found, we break from the loop. The loop counter variable (j) will describe the index of this property group within the CPropertyGroup array.

```
        // Determine if we already have a property group for this material
        for ( j = 0; j < pLightGroup->m_nPropertyGroupCount; j++ )
        {
            // Break if material index matches
            if ( (long)pLightGroup->m_pPropertyGroup[j]->m_nPropertyData == MaterialIndex )
               break;
        }
```

If the loop counter (j) is equal to the number of property groups that the light group contains, it means that a property group could not be found that already uses the current material. If this is the case then we need to add a new property group to the CPropertyGroup array that will use this material:

```
        // If we didn't have this property group, add it
        if ( j == pLightGroup->m_nPropertyGroupCount )
        {
            if ( pLightGroup->AddPropertyGroup( ) < 0 ) goto BuildFailure;

            // Set up new property group data
            pProperty = pLightGroup->m_pPropertyGroup[ j ];
            pProperty->m_PropertyType  = CPropertyGroup::PROPERTY_MATERIAL;
            pProperty->m_nPropertyData = (ULONG)MaterialIndex;
            pProperty->m_nVertexStart  = pLightGroup->m_nVertexCount;
            pProperty->m_nVertexCount  = 0;
         }
```

This is an important piece of code because it is the only place where a new property group gets created. Remember, the first time this function is called by ProcessMeshes there will be no property groups and no light groups. These will be created as ProcessMeshes calls this function once for every material used by the scene.

In the above code we have used CLightGroup::AddPropertyGroup to resize the CPropertyGroup array so that there is space at the end for a new property group. We assign the property group the PROPERTY_MATERIAL property type so that we know this is a material property and we store the material index that this property group is mapped to in the *m_nPropertyData* member. Finally, we

record the current number of vertices that are in its parent light group vertex array in the *m_nVertexStart* member. This is important because this is where the faces for this property group will start in the vertex buffer and will be used during the DrawIndexedPrimitive function so that the pipeline knows to add this amount to each index belonging to this property group. This works because the BuildLightGroups function is called one per material from the ProcessMeshes function. If 10 faces in the passed vector belong to this property group, they will all have their vertices copied into the vertex buffer together.

Section 3 : Adding Vertices to Light Groups and Indices to Property Groups

At this point in the function we have a pointer to the light group and the property group to which the face belongs. We call CScene::ProcessVertices to copy the vertices of the current face into the light group vertex buffer. We also call CScene::ProcessIndices to copy or create the indices that will be placed into the property group index buffer.

```
        // Process the vertices / indices and store in this property group
        pProperty = pLightGroup->m_pPropertyGroup[ j ];
        if (!ProcessIndices( pLightGroup, pProperty, pSurface ) ) return false;
        if (!ProcessVertices( pLightGroup, pProperty, pSurface ) ) return false;

    } // Next Surface
```

As you can see, this process occurs for every surface passed into the function in the STL vector. Remember that all of the surfaces in that vector share the same material. This is how we make sure that faces are being placed into the various light group vertex buffers in material order.

All of the surfaces passed in have now been assigned to light groups, so we can delete the LightContribution array and the SelectedLights array since we no longer need them. Finally, we return success.

```
    // Release memory
    if ( LightContribution ) delete []LightContribution;
    if ( SelectedLights ) delete []SelectedLights;

    // Success!
    return true;
```

The BuildFailure label can be jumped to from several places in the code if a memory allocation fails. Using a goto command prevents us having to duplicate memory release code in several places if something goes wrong.

```
BuildFailure:
    // If we dropped here, something bad happened :)
    if ( LightContribution ) delete []LightContribution;
    if ( SelectedLights ) delete []SelectedLights;

    // Failure!
    return false;
}
```

CScene::GetLightContribution

This function accepts a light and a surface and returns an influence score for that light with regards to the surface.

A naïve first approach might be to base the score on the distance from each vertex in the surface to the light source and average them. This is not a good idea. Even if a light is extremely close to a vertex it does not necessarily mean the light has a large influence on the resulting color of the vertex. The light may be extremely dim such that a much brighter light source further away would contribute more color to the vertex.

It would seem the only way we can find out for sure what influence the light will have is if we use it to light the vertices in the surface and then examine the resulting color. So how exactly do we do this?

In our textbook, we examined the lighting calculations that are performed by the pipeline. So all we have to do is emulate that model and we will have the final vertex color for every vertex in the face. This involves calculating the amount of color that reaches each vertex from the light source and then modulating this color with the material members of the material assigned to the surface. At this point, we have the final color of the vertices. Remember we are doing this for a single light source only in this function, so the color/intensity of each vertex directly describes the amount of color contributed by this light source only.

At this point, we could just add up the RGB components of the brightest vertex and return this as the score, but this would not be successful in all situations. Imagine for example a vertex color of RGB (0.5, 0.5, 0.5) which is a half intensity light (a gray light). This would have a combined score of (0.5 + 0.5 + 0.5) = 1.5. Now this is not a particularly brightly lit vertex, whereas a vertex color of (1.0, 0.0, 0.0) would have a lower score but have full intensity red. This is something we want to watch out for because if we place a full intensity red light in our scene, it would only affect the red color component of the vertices it lights. Therefore, what we will do is use the highest color component (R, G, or B) of the vertex as the vertex score and return the highest vertex score found for the surface.

We must also make sure that we do not take sign into account when doing this scoring because as we mentioned earlier, it is possible to place dark lights in the scene which have negative color emitting properties. These lights detract light color from the vertices they influence and since we are doing this for only a single light, this would result in negative RGB values for the vertices in the surface. But this does not mean that the light is any less important in determining the final color of the vertex, so we must make this an absolute value comparison.

Once we have collected the highest R, G, or B component from the vertices of the face, it is this highest color component that is returned to the calling function (BuildLightGroups) and entered into the LightContribution table.

The overall process looks like this:

- Best Score = 0
- For each vertex in passed surface
- Calculate the diffuse color emitted from the light and modulate it with the material diffuse property
- Calculate the ambient color emitted from the light and modulate it with the material ambient property
- Add these colors together to get the overall color of the vertex received from this light source.
- Find which color component is higher for this vertex and if this is higher than the highest color component found from previous vertices in the loop, make this the new best score
- Return the best score

Notice how we only calculate the diffuse and ambient contributions of the light source and not the specular. This is because specular lighting is camera position dependant and as the camera position constantly changes, it will not accurately describe the importance of that light to a vertex. However given this arrangement it is important to note that if you do wish specular lights to exist in the scene, they must not be separate light objects. They should be included with a light that has a diffuse and/or ambient color source as well so that they are not ignored by this process.

The nice thing about this code is that it gives us additional insight into how the DirectX pipeline calculates vertex colors. This could be very handy if you are not using DirectX lighting but are instead storing the vertex colors within each vertex (pre-lit vertices). You could use a function similar to this one as a pre-process to generate your vertex colors such that it looks like they are being lit by the DirectX pipeline**. There is a real benefit to doing this because it relieves the pipeline from having to perform lighting calculations at runtime.

The Lighting Calculations Revisited

The vertex coloring process can be described as follows:

VertexDiffuse = MaterialDiffuseColor * LightDiffuseColor * Dot * AttFactor * SpotFactor.
VertexAmbient = MaterialAmbientColor * LightAmbientColor * AttFactor * SpotFactor
VertexColor = VertexDiffuse + VertexAmbient

*** we are not taking specular color into account here*
MaterialDiffuseColor – This is the diffuse reflectance property of the material. It is an RGB color describing how to scale the RGB components of incoming diffuse light.

MaterialAmbientColor – This is the ambient reflectance property of the material. It is an RGB color describing how to scale the RGB components incoming ambient light.

LightDiffuseColor – This is the diffuse color of the light source.

LightAmbientColor – This is the ambient color of the light source.

Dot – This is the result of the dot product between the vertex normal and the vector from the vertex to the light source. As both of these vectors are unit length, this equates to the cosine of the angle between these two vectors and will be between 0.0 and 1.0. This is used to scale the result of *MaterialDiffuseColor*LightDiffuseColor* in the above diffuse equation to take the orientation between the vertex and the light into account.

AttFactor – For directional light types this will always be 1.0 and will not scale the color in any way. For spot lights and point lights, the AttFactor is the result of the attenuation equation discussed earlier and shown below. This value will be between 0.0 and 1.0 and is used to scale the color based on how the light attenuates as distance to the vertex from the light increases.

$$\text{AttFactor} = \frac{1}{Attenuation1 + (Attenuation2 \times D) + (Attenuation3 \times D^2)}$$

Attenuation1, Attenuation2 and Attenuation3 are the attenuation settings of the light source and D is the distance from the light source to the vertex.

SpotFactor – If the light is a directional light or a point light then this value will always be set to 1.0. The SpotFactor is used to further scale the color only when a spot light is being processed. The spot factor, which will be between 0.0 and 1.0, is used to scale the color based on the position of the vertex between the inner and outer cones of the spot light.

We calculate the dot product of the vertex normal and a vector from the vertex to the light. This gives us the cosine of the angle between them and is used to determine which cone the vertex falls into. If

the angle is smaller than the inner cone angle/2 then the vertex is within the inner cone and the SpotFactor should be set to 1.0. If the angle is larger than the outer cone angle/2 then the vertex is completely outside the cones of influence of the spot light and so the SpotFactor should be set to 0.0. Otherwise the vertex is located between the inner and outer cone and we calculate the SpotFactor using the following equation.

α = Angle between Vertex Normal and VertexToLight direction vector

ϕ = Phi / 2 (Half the outer cone angle)

θ = Theta /2 (Half the inner cone angle)

Falloff = Falloff property of the **D3DLIGHT9** structure

$$\text{SpotFactor} = \left(\frac{\cos(\alpha) - \cos(\phi)}{\cos(\theta) - \cos(\phi)} \right)^{Falloff}$$

Now it is time to put all of this into code. We break things up one section at a time to make for easier reading:

```
float CScene::GetLightContribution( iwfSurface * pPoly, D3DLIGHT9 * pLight )
{
    D3DXVECTOR3 Direction, LightDir = pLight->Direction;
    float       Contribution = 0.0f, MaxContribution = 0.0f;
    D3DXCOLOR   Diffuse, Ambient, Color;
    float       Atten, Spot, Rho, Dot;
    float       Distance;
    ULONG       i;

    // We can only get light contribution of we have a material
    if ( pPoly->ChannelCount == 0 || !(pPoly->Components & SCOMPONENT_MATERIALS ))
        return 0.0f;
    if ( pPoly->MaterialIndices[0] < 0 )
        return 0.0f;
```

The first thing we do is check to see if the passed surface has a material applied. If not, then this surface cannot reflect light and as such this light will have no influence on the face. It is possible for surfaces not to have materials but this is never the case with IWF files exported from GILES.

Next we retrieve the material that this surface uses from the CScene::m_pMaterialLight array. It is possible that a surface may have multiple channel counts, but GILES only allows a single material per surface, so the material index will always be stored in array element zero.

```
    // Retrieve the material for colour calculations
    D3DMATERIAL9 * pMaterial = &m_pMaterialList[ pPoly->MaterialIndices[0] ];
```

Now we will loop through each vertex in the face to calculate their colors. We start be calculating the light direction vector (the vector from the vertex to the light). We also record the length of this vector which will tell us the distance from the light to the vertex. We use this to check if the vertex is within range of the light. If it is not, then we can skip this vertex because it is not influenced by this light. Notice that we only skip an out of range vertex if the light we are processing is not a directional light

because directional lights have infinite range. After that, we normalize the light direction vector so that it is ready to use later when we perform the dot product with it and the vertex normal.

```
    // Loop through each vertex
    for ( i = 0; i < pPoly->VertexCount; i++ )
    {
        // Retrieve lighting formula params
        Direction = (D3DXVECTOR3&)pPoly->Vertices[i] - pLight->Position;
        Distance  = D3DXVec3Length( &Direction );

        // Skip if the light is out of range of the vertex (does not apply to directional)
        if ( pLight->Type != D3DLIGHT_DIRECTIONAL && Distance > pLight->Range ) continue;

        // Normalize our direction from the vertex to the light
        D3DXVec3Normalize( &Direction, &Direction );
```

Now we will calculation the attenuation factor using the attenuation values stored in the light and the distance from the light to the vertex we calculated above. We initially set the attenuation factor to 1.0 and skip the calculation if this is a directional light because directional lights have infinite range and do not attenuate with distance.

```
      // Calculate light's attenuation factor.
      Atten = 1.0f;
      if ( pLight->Type != D3DLIGHT_DIRECTIONAL )
      {
          Atten = ( pLight->Attenuation0 + pLight->Attenuation1 * Distance
                    + pLight->Attenuation2 * (Distance * Distance));
          if ( Atten > 0 ) Atten = 1 / Atten; // Avoid divide by zero case
      } // End if not a directional light
```

Now we calculate the SpotFactor -- this is 1.0 for any other light type except spot lights. First we calculate the cosine of the angle between the light direction vector and the vertex normal by performing a dot product between them (Rho).

```
        // Calculate light's spot factor
        Spot = 1.0f;
        if ( pLight->Type == D3DLIGHT_SPOT ){
            // Calculate RHO
            Rho = fabsf(D3DXVec3Dot( &(-LightDir), &Direction ));
```

Next we compare this angle with the cosine of half the angle of the inner cone. We use half the inner cone angle because the light center runs down the middle of the cone with an angle theta/2 on either side of this center point. If Rho is larger than the cosine of the half angle, then the vertex is inside the inner cone and should not have falloff applied (we set the spot factor to 1.0). If Rho is smaller than half the outer cone angle then the vertex is completely outside the outer cone and is not influenced by the light (spot factor should be set to 0).

```
            if ( Rho > cosf( pLight->Theta / 2.0f ) )
                Spot = 1.0f;
            else if ( Rho <= cosf( pLight->Phi  / 2.0f ) )
                Spot = 0.0f;
```

If none of the above cases are true, then we calculate the falloff using the spot factor equation shown above. In this case the vertex falls in the space between the inner and outer cones of the spot light.

```
            else
            {
                Spot = ((Rho - cosf( pLight->Phi / 2.0f)) / (cosf( pLight->Theta / 2.0f)
                       - cosf( pLight->Phi / 2.0f))) * pLight->Falloff;
            }
        } // End if Spotlight
```

At this point we have the spot factor and the attenuation factor, so all that is let to do is to calculate the diffuse color and ambient colors reflected by the material and store them in the vertex.

Diffuse lighting must be scaled by the cosine of the angle between the vertex normal and the light direction vector, so we take the dot product between them, which we will refer to as the dot factor. Note that for diffuse lights we must take account of vertices that are facing away from the light source – they should not receive any diffuse light.

```
        Dot = D3DXVec3Dot( (D3DXVECTOR3*)&pPoly->Vertices[i].Normal, &Direction );
        if(Dot <= 0) Dot = 0;
```

Now we can multiply the material diffuse color with the light diffuse color and scale all of this by the dot, attenuation, and spot factors as shown below. Notice that we ignore the alpha color component as it is not used for lighting.

```
        // Calculate diffuse contribution for this vertex (Cd*Ld*(N.Ldir)*Atten*Spot)
        Diffuse.a = 0;
        Diffuse.r = pMaterial->Diffuse.r * pLight->Diffuse.r * Dot * Atten * Spot;
        Diffuse.g = pMaterial->Diffuse.g * pLight->Diffuse.g * Dot * Atten * Spot;
        Diffuse.b = pMaterial->Diffuse.b * pLight->Diffuse.b * Dot * Atten * Spot;
```

We now calculate the ambient contribution of the light source in exactly the same way with the exception that the dot factor is not used because ambient light is not orientation dependant.

```
        // Calculate ambient contribution for this vertex (Ca*[Ga + sum(Lai)*Atti*Spoti])
        Ambient.a = 0;
        Ambient.r = pMaterial->Ambient.r * (  pLight->Ambient.r * Atten * Spot );
        Ambient.g = pMaterial->Ambient.g * (  pLight->Ambient.g * Atten * Spot );
        Ambient.b = pMaterial->Ambient.b * (  pLight->Ambient.b * Atten * Spot );
```

We need to add these two colors together (diffuse and ambient) to get the final vertex color as received from this light source.

```
        Color = Ambient + Diffuse;
```

To get the highest color component of this new color we perform comparisons using the fabs function so that negative numbers still have weight.

```
        // Calculate light contribution (fabsf() because even dark-lights contribute)
        Contribution = fabsf(Color.r);
        if ( fabsf(Color.g) > Contribution ) Contribution = fabsf(Color.g);
        if ( fabsf(Color.b) > Contribution ) Contribution = fabsf(Color.b);
```

With the highest color component for this vertex found, we check it against the current highest component found so far. If the new color is higher, this becomes the new high score.

```
        // Store the maximum contribution to this surface.
        if ( Contribution > MaxContribution )  MaxContribution = Contribution;

    } // Next Vertex
```

By the time we have done the above for each vertex in the surface, the local variable MaxContribution will contain the highest single color component calculated for any of the surfaces vertices. We return this value which will become this light's score in the light contribution table for this surface.

```
    // Return the total contribution this light gives to face
    // This will be put in the light contribution table.
    return MaxContribution;
 }
```

We call this function for each light in the scene once for every surface that we process. This means that we build a light score table one surface at a time. The scores in this table we have just discovered indicate the highest color component that was calculated for the surface vertices using the relevant light.

CScene::ProcessVertices

The ProcessVertices function takes the vertices in the passed surface and adds them to the light group CVertex array. First the function stores the current vertex count so that it knows the array element where the new vertices will be placed. Then the function calls CLightGroup::AddVertex to resize the CVertex array to make room for the new vertices. We then loop through every vertex in the passed surface and copy them into the CVertex array.

```
bool CScene::ProcessVertices( CLightGroup * pLightGroup, CPropertyGroup *pProperty,
                              iwfSurface * pFilePoly )
{
    ULONG i, VertexStart = pLightGroup->m_nVertexCount;

    // Allocate enough vertices
    if ( pLightGroup->AddVertex( pFilePoly->VertexCount ) < 0 ) return false;

    // Loop through each vertex and copy required data.
    for ( i = 0; i < pFilePoly->VertexCount; i++ )
    {
        // Copy over vertex data
        pLightGroup->m_pVertex[i + VertexStart].x      = pFilePoly->Vertices[i].x;
        pLightGroup->m_pVertex[i + VertexStart].y      = pFilePoly->Vertices[i].y;
```

```
        pLightGroup->m_pVertex[i + VertexStart].z      = pFilePoly->Vertices[i].z;
        pLightGroup->m_pVertex[i + VertexStart].Normal =
                                            (D3DXVECTOR3&)pFilePoly->Vertices[i].Normal;
        pProperty->m_nVertexCount++;

    } // Next Vertex

    // Success!
    return true;
}
```

Because this function is called from BuildLightGroups (on a per material basis), vertices will always be added to groups with other vertices that use the same materials.

CScene::ProcessIndices

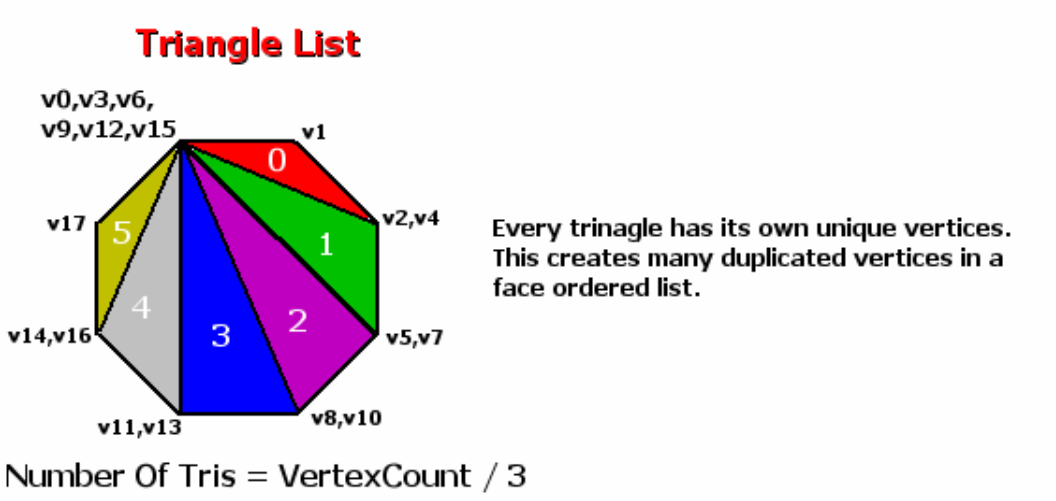
The ProcessIndices function is called from the BuildLightGroups function once we have located a property group that has the same material as the face. The surface passed in has to have its indices added to the index array (there may be other indices from other faces already stored). This index array will later be copied into the property group index buffer using the BuildBuffers function.

There is one problem to consider: in some cases, the surface passed into this function may not have indices pre-generated for it or even if it does, they may be stored as strips or fans. Bearing this in mind, this function has to allow for these cases. If the surface passed into the function does not contain indices, then we will need to generate an indexed triangle list ourselves. If the surface does have indices and they are arranged as an index triangle list, then we can copy them directly into the index array. However, if the indices in the surface are not in indexed triangle list format (indexed strips or indexed fans) we will need to convert these indices into an index triangle list before we add them to the array.

Before we cover the code to the function itself, we need to look at the different ways a surface can be stored and discuss what is involved in generating an indexed triangle list for it. We will first cover the case of surfaces that do not include indices. They will have an ordered vertex list and the vertices may be stored in triangle list, triangle strip, or triangle fan format.

Triangle List to Indexed Triangle List

If a surface is stored as a triangle list, it means that are three unique vertices for each triangle in the surface. If we have a surface made from 6 triangles, there will be exactly 18 (6*3) vertices. Generating an indexed triangle list for this type of surface is easy because the numbers of the vertices in the array forming each triangle are the actual index numbers themselves. The following picture shows an octagon surface stored as a triangle list:



In this example there is one position in all faces that is duplicated into 6 different vertices (v1,v3,v6,v9,v12,15). The vertex list for this surface would be stored as follows:

VertexList = v0,v1,v2 , v3,v4,v5 , v6,v7,v8 , v9,v10,v11 , v12,v13,v14 , v15,v16,v17

If we encounter one of these surfaces, generating the indices for each triangle is as easy as using the vertex numbers for each face. In other words, the index list describing triangles 0 through 5 would look like so:

Index List { 0,1,2 , 3,4,5 , 6,7,8 , 9,10,11 , 12,13,14 , 15,16,17 }

It should be clear that the number of indices we need is equal to the number of vertices in the surface. So the code for generating the indexed triangle list and adding it to the property group index buffer would look like so:

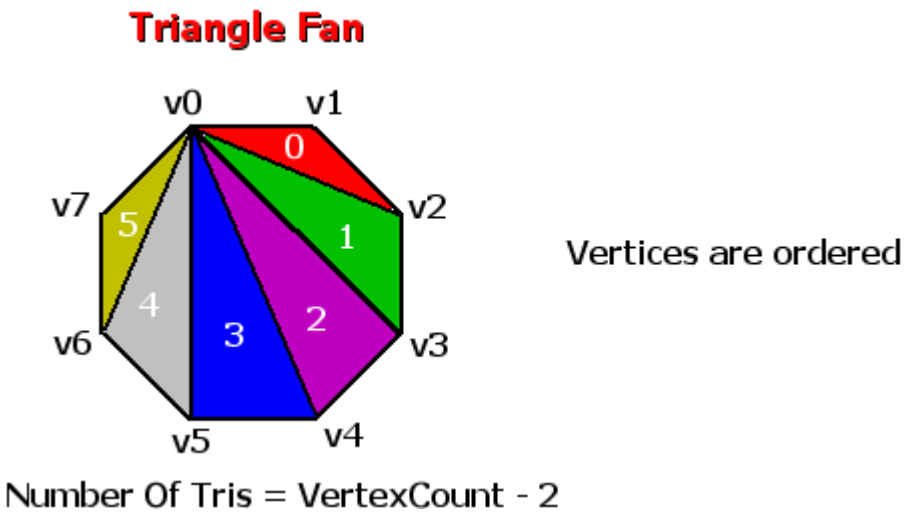
```
VertexStart = pLightGroup->m_nVertexCount - pProperty->m_nVertexStart;
IndexCount  = pProperty->m_nIndexCount;

case VERTICES_TRILIST:

  // Resize the index array large enough to hold the indices we are about to create.
  // The count is equal to the vertex count
  if( pPropertyGroup->AddIndex( pFace->VertexCount ) < 0 )
  for( i = 0; i < pFace->VertexCount; i++ )
  {
       pPropertyGroup->m_pIndex[i + IndexCount] = i + VertexStart;
  }
 break;
```

When the passed surface is a triangle list we call the AddIndex function to resize the property group index array using the vertex count of the surface. Next, we loop through each vertex in the face and the element number of the vertex in the array becomes the index in the index list.

Note: IndexCount is the number of indices in the index array prior to adding this face's indices. It ensures that we add our indices to the end of the index list. Vertex Start is the position in the light group vertex buffer where the first vertex used by this property group is. We do this because all indices stored in a surface start at 0 and are local to the face. Since many faces may exist in this index buffer we must add the number of vertices already being used by this property group onto the index of each face added so that the index is no longer relative to the surface but to the entire index buffer to which it is being added.

Triangle Fan to Indexed Triangle List

If the surface is a triangle fan, then the vertices will be ordered in a clockwise winding order and there will be no duplicated vertices. No index list will exist so we will have the generate one. Because we use the first vertex (v0) for every face, the code for generating the indexed triangle list is straightforward.



We need three indices per face and we know the triangle count of a triangle fan is VertexCount-2. Therefore, if we multiply this by 3 we have the correct number of indices for every triangle in the face. In the above example you can see that the triangle count is:

VertexCount-2 = 8 – 2 = 6 Triangles

6*3 = 18 indices required

The following code shows that if we use the number of the first vertex in vertex list in every triangle, then generating the indices can be done by stepping through the number of faces, each time incrementing the other two vertices. For example, 0,1,2 followed by 0,2,3 followed by 0,3,4 and so on.

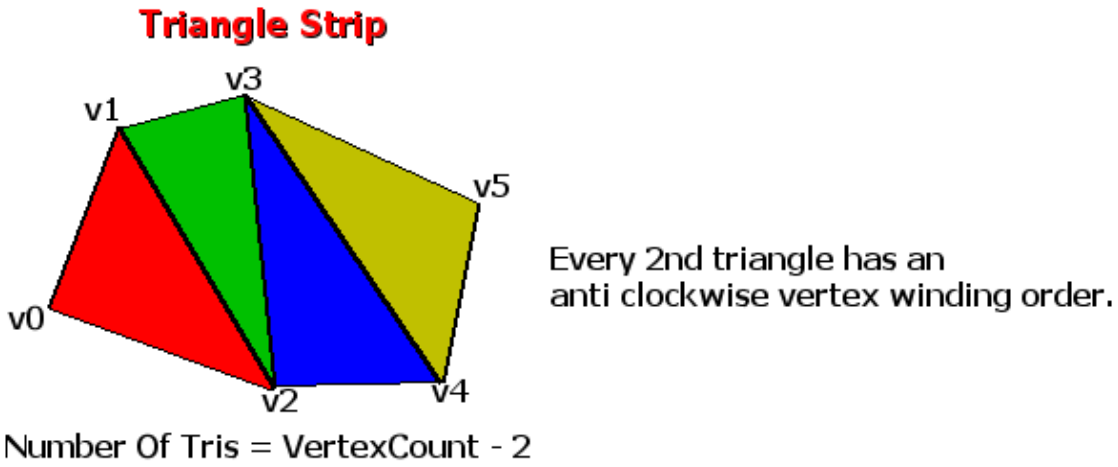
```
VertexStart = pLightGroup->m_nVertexCount - pPropertyGroup->m_nVertexStart;
IndexCount  = pPropertyGroup->m_nIndexCount;
```

```
case VERTICES_TRIFAN:

     // Allocate space in the property groups index array (NumberOfTris * 3)
     if ( pPropertyGroup->AddIndex( (pFace->VertexCount - 2 ) * 3 ) < 0 ) ;

     for ( Counter = IndexCount, i = 1; i < pFace->VertexCount - 1; i++ )
     {
           pPropertyGroup->m_pIndex[ Counter++ ] = VertexStart;
           pPropertyGroup->m_pIndex[ Counter++ ] = i + VertexStart;
           pPropertyGroup->m_pIndex[ Counter++ ] = i + 1 + VertexStart;
     } // Next triangle
break;
```

Note: In the code above, IndexCount is the number of indices in the property group prior to this surface's indices being added -- it is the index where the new values should be added in the array.

Triangle Strip to Indexed Triangle List

If the surface is stored in triangle strip format, then there will be no duplicated vertices and the vertex list will be ordered to describe a continuous set of triangles. The number of triangles created by the list of vertices is calculated as *VertexCount-2*. We multiply this by 3 to get the number of indices we need to describe this surface as an indexed triangle list.



In Chapters Two and Three we examined the culling order requirements for triangle strips. We know that DirectX expects every second face to have a counter-clockwise winding order in order to render a strip correctly. You can see in the above image that the second triangle (v1,v2,v3) and the fourth triangle (v3,v4,v5) meet this standard. To ensure that indexed triangle list rendering does not cull these faces, we must flip the order of every second triangle in the surface so that their indices are clockwise.

```
case VERTICES_TRISTRIP:

       // Allocate index we need (NumberOfTris * 3)
```

```
       if ( pPropertyGroup->AddIndex( (pFace->VertexCount - 2) * 3 ) < 0 ) ;
       for ( Counter = IndexCount, i = 0; i < pFace->VertexCount - 2; i++ )
       {
           // Starting with triangle 0.
           // Is this an 'Odd' or 'Even' triangle
           if ( (i % 2) == 0 )
             {
                pProperty->m_pIndex[ Counter++ ] = i + VertexStart;
                pProperty->m_pIndex[ Counter++ ] = i + 1 + VertexStart;
                pProperty->m_pIndex[ Counter++ ] = i + 2 + VertexStart;
              }
           else
             {
                pProperty->m_pIndex[ Counter++ ] = i + VertexStart;
                pProperty->m_pIndex[ Counter++ ] = i + 2 + VertexStart;
                pProperty->m_pIndex[ Counter++ ] = i + 1 + VertexStart;
             }
       } // Next vertex

break;
```

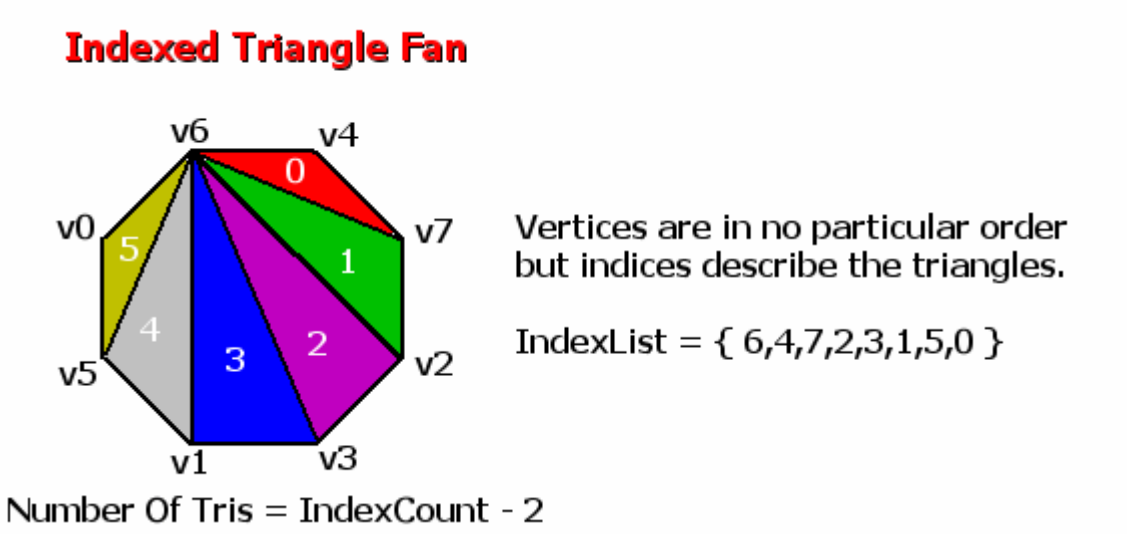
Indexed Triangle List to Indexed Triangle List

If the surface we are assigning to this property group already has an array of indices in indexed triangle list format, we can simply copy the indices from the surface straight into the index array as shown below. We must remember to add on the VertexStart value to each index to be sure that the index values are relative to the first vertex used by the property group in the light group vertex buffer.

```
case INDICES_TRILIST:

      // We can do a straight copy (converting from 32bit to 16bit if necessary)
      if ( pPropertyGroup->AddIndex( pPropertyGroup->IndexCount ) < 0 )
     for ( i = 0; i < pFace->IndexCount; i++ )
     {
          pPropertyGroup->m_pIndex[i + IndexCount] = pFace->Indices[i] + VertexStart;
     }
break;
```

Indexed Triangle Fan to Indexed Triangle List

When the imported surface is an indexed triangle fan, the vertices are not guaranteed to be in any specific order but the list of indices accompanying the surface describes the correct vertices to form a triangle fan. The number of triangles in the face can be calculated as IndexCount - 2.

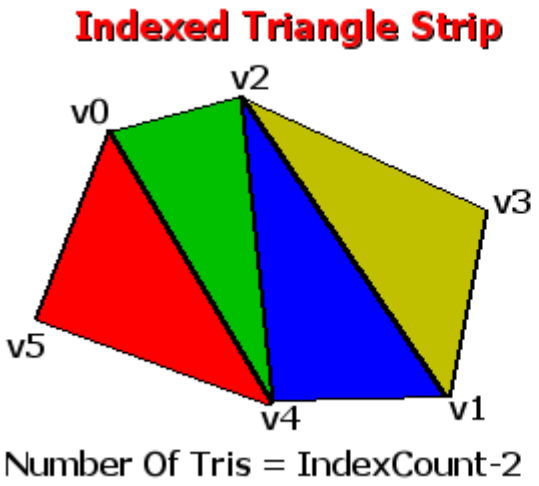


Calculating the triangle list indices that we need for our index buffer is almost exactly the same as the way we did it for the non-indexed triangle fan case:

```
if ( pPropertyGroup->AddIndex( (pFace->IndexCount - 2 ) * 3 ) < 0 );
for ( Counter = IndexCount, i = 1; i < pFace->VertexCount - 1; i++ )
{
       pPropertyGroup->m_pIndex[ Counter++ ] = pFace->Indices[ 0 ] + VertexStart;
       pPropertyGroup->m_pIndex[ Counter++ ] = pFace->Indices[ i ] + VertexStart;
       pPropertyGroup->m_pIndex[ Counter++ ] = pFace->Indices[ i + 1 ] + VertexStart;
}
```

Instead of using the loop variable *i* as the index to the vertex as in the non-indexed case, we use it to index into the surface index list to return the number of the vertex at that index. It is this value that is used to build the indices of our triangle list.

Indexed Triangle Strip to Indexed Triangle List



Vertices in no particular order but indices describe the strip

Index Buffer = { 5,0,4,2,1,3 }

With an indexed triangle strip, the vertices not guaranteed to be in any particular order because the index list is responsible for describing the vertices in strip format. When using an indexed triangle strip, the number of indices is equal to the number of vertices, and the number of triangles created by the strip is calculated as IndexCount - 2. In the above example 6 - 2 = 4 is correct because there are four triangles.

```
case INDICES_TRISTRIP:

        if ( pPropertyGroup->AddIndex( (pFace->IndexCount - 2) * 3 ) < 0 ) ;
        for ( Counter = IndexCount, i = 0; i < pFace->IndexCount - 2; i++ )
        {
              // Starting with triangle 0.
              // Is this an 'Odd' or 'Even' triangle
              if ( (i % 2) == 0 )
              {
                   pPropertyGroup->m_pIndex[Counter++] = pFace->Indices[i] + VertexStart;
                   pPropertyGroup->m_pIndex[Counter++] = pFace->Indices[i+1] + VertexStart;
                   pPropertyGroup->m_pIndex[Counter++] = pFace->Indices[i+2] + VertexStart;
              }
              else
              {
                   pPropertyGroup->m_pIndex[Counter++] = pFace->Indices[i]   + VertexStart;
                   pPropertyGroup->m_pIndex[Counter++] = pFace->Indices[i+2] + VertexStart;
                   pPropertyGroup->m_pIndex[Counter++] = pFace->Indices[i+1] + VertexStart;
              }
        } // Next vertex

break;
```

Below you can see the code to the ProcessIndices function in its entirety. It includes all of the code snippets we have just covered above. We pass in the surface that is about to have its indices added to

the property group, a pointer to the property group itself, and the light group to which the property group belongs.

```
bool CScene::ProcessIndices(CLightGroup *pLightGroup, CPropertyGroup *pProperty,
                            iwfSurface * pFilePoly )
{
    ULONG i, Counter, VertexStart, IndexCount;

    VertexStart = pLightGroup->m_nVertexCount - pProperty->m_nVertexStart;
    IndexCount  = pProperty->m_nIndexCount;

    // Generate indices
    if ( pFilePoly->IndexCount > 0 )
    {
        ULONG IndexType = pFilePoly->IndexFlags & INDICES_MASK_TYPE;

        // Interpret indices (we want them in tri-list format)
        switch ( IndexType )
        {
            case INDICES_TRILIST:

                // We can do a straight copy (converting from 32bit to 16bit)
                if ( pProperty->AddIndex( pFilePoly->IndexCount ) < 0 ) return false;
                for ( i = 0; i < pFilePoly->IndexCount; i++ )
                     pProperty->m_pIndex[i + IndexCount] = \
                                              pFilePoly->Indices[i] + VertexStart;
                break;

            case INDICES_TRISTRIP:

                // Index in strip order
                if ( pProperty->AddIndex( (pFilePoly->IndexCount - 2) * 3 ) < 0 )
                   return false;
                for ( Counter = IndexCount, i = 0; i < pFilePoly->IndexCount - 2; i++ )
                {
                    // Starting with triangle 0.
                    // Is this an 'Odd' or 'Even' triangle
                    if ( (i % 2) == 0 )
                    {
                        pProperty->m_pIndex[Counter++] = \
                                                  pFilePoly->Indices[i] + VertexStart;
                        pProperty->m_pIndex[Counter++] = \
                                                  pFilePoly->Indices[i + 1] + VertexStart;
                        pProperty->m_pIndex[Counter++] = \
                                                  pFilePoly->Indices[i + 2] + VertexStart;

               } // End if 'Even' triangle
               else
               {
                   pProperty->m_pIndex[Counter++]= pFilePoly->Indices[i] + VertexStart;
                   pProperty->m_pIndex[Counter++]= pFilePoly->Indices[i + 2] + VertexStart;
                   pProperty->m_pIndex[Counter++]= pFilePoly->Indices[i + 1] + VertexStart;

               } // End if 'Odd' triangle

            } // Next vertex

            break;
```

```
            case INDICES_TRIFAN:

            // Index in fan order.
            if ( pProperty->AddIndex( (pFilePoly->IndexCount - 2 ) * 3 ) < 0 )
               return false;

            for ( Counter = IndexCount, i = 1; i < pFilePoly->VertexCount - 1; i++ )
            {
              pProperty->m_pIndex[ Counter++ ] = pFilePoly->Indices[ 0 ] + VertexStart;
              pProperty->m_pIndex[ Counter++ ] = pFilePoly->Indices[ i ] + VertexStart;
              pProperty->m_pIndex[ Counter++ ] = pFilePoly->Indices[ i + 1 ] + VertexStart;
            } // Next Triangle

            break;

        } // End Switch

    } // End if Indices Stored
    else
    {
        // We are going to try and build the indices ourselves
        ULONG VertexType = pFilePoly->VertexFlags & VERTICES_MASK_TYPE;

        // Interpret vertices (we want our indices in tri-list format)
        switch ( VertexType )
        {
            case VERTICES_TRILIST:

                // Straight fill
                if ( pProperty->AddIndex( pFilePoly->VertexCount ) < 0 ) return false;
                for ( i = 0; i < pFilePoly->VertexCount; i++ )
                     pProperty->m_pIndex[i + IndexCount] = i + VertexStart;

                break;

            case VERTICES_TRISTRIP:

                // Index in strip order
                if ( pProperty->AddIndex( (pFilePoly->VertexCount - 2) * 3 ) < 0 )
                    return false;
                for ( Counter = IndexCount, i = 0; i < pFilePoly->VertexCount - 2; i++ )
                {
                    // Starting with triangle 0.
                    // Is this an 'Odd' or 'Even' triangle
                    if ( (i % 2) == 0 )
                    {
                        pProperty->m_pIndex[ Counter++ ] = i + VertexStart;
                        pProperty->m_pIndex[ Counter++ ] = i + 1 + VertexStart;
                        pProperty->m_pIndex[ Counter++ ] = i + 2 + VertexStart;

                    } // End if 'Even' triangle
                    else
                    {
                        pProperty->m_pIndex[ Counter++ ] = i + VertexStart;
                        pProperty->m_pIndex[ Counter++ ] = i + 2 + VertexStart;
                        pProperty->m_pIndex[ Counter++ ] = i + 1 + VertexStart;

                    } // End if 'Odd' triangle
```

```
                } // Next vertex

                break;

            case VERTICES_TRIFAN:

                // Index in fan order.
                if ( pProperty->AddIndex( (pFilePoly->VertexCount - 2 ) * 3 ) < 0 )
                    return false;
                for ( Counter = IndexCount, i = 1; i < pFilePoly->VertexCount - 1; i++ )
                {
                    pProperty->m_pIndex[ Counter++ ] = VertexStart;
                    pProperty->m_pIndex[ Counter++ ] = i + VertexStart;
                    pProperty->m_pIndex[ Counter++ ] = i + 1 + VertexStart;

                } // Next Triangle

                break;

        } // End Switch

    } // End if no Indices stored

    // Success!
    return true;
}
```

CScene::Render

To render the scene we must traverse each light group, set up its lights and its vertex buffer, and then loop through each of the property groups, set the material and index buffer for the property group and call DrawIndexedPrimitive.

```
void CScene::Render( )
{
    ULONG         i, j;
    CLightGroup * pLightGroup = NULL;
    ULONG       * pLightList  = NULL;

    // Set up our dynamic lights
    m_pD3DDevice->SetLight( 0, &m_DynamicLight );
    m_pD3DDevice->LightEnable( 0, TRUE );
```

In the above section we set our dynamic light in the reserved device light slot 0 and enable it. We set the light each frame because the light may be animated and we want to make the device aware of the new settings.

Now, we loop through each light group in the CScene light group array and get a pointer to both the current light group and the light group's light list:

```
    // Loop through each light group
    for ( i = 0; i < m_nLightGroupCount; i++ )
    {
        // Set active lights
        pLightGroup = m_ppLightGroupList[i];
        pLightList  = pLightGroup->m_pLightList;
```

Since dynamic lights will be using the lower device slot values, we just count up from the number of reserved lights slots allocated by the application to the maximum simultaneous light limit, and set and enable the lights of our light group. First we check if the light slot we are processing has a light in the current light group that should go there and if not, we disable any lights there. This is important because if the light group before used eight lights and the current group only uses four, there will be lights that remain active that we will want disabled before rendering the current polygon batch.

```
        for ( j = m_nReservedLights; j < m_nLightLimit; j++ )
        {
            if ( (j - m_nReservedLights) >= (pLightGroup->m_nLightCount ) )
            {
                // Disable any light sources which should not be active
                m_pD3DDevice->LightEnable( j, FALSE );

            } // End if no more lights
            else
            {
                // Set this light as active
                m_pD3DDevice->SetLight( j,
                                        &m_pLightList[pLightList[j - m_nReservedLights]]);
                m_pD3DDevice->LightEnable( j, TRUE );

            } // End if set lights

        } // Next Light
```

Once we have setup all the lights for the current light group, we set the light group vertex buffer as the current vertex stream.

```
        // Set vertex stream
        m_pD3DDevice->SetStreamSource( 0, pLightGroup->m_pVertexBuffer,
                                       0, sizeof(CVertex));
```

Next we loop through the light group property group array. We retrieve the material from the current property group and set it as the device material. Next we set the index buffer of the property group as the current device index buffer. Finally we call the DrawIndexedPrimitive function to render the triangles in the property group index buffer.

```
        // Now loop through and render the associated property groups
        for ( j = 0; j < pLightGroup->m_nPropertyGroupCount; ++j )
        {
            CPropertyGroup * pProperty = pLightGroup->m_pPropertyGroup[j];
            m_pD3DDevice->SetMaterial(&m_pMaterialList[(long)pProperty->m_nPropertyData] );
            m_pD3DDevice->SetIndices(pProperty->m_pIndexBuffer);
            m_pD3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST,
```

```
                                                pProperty->m_nVertexStart, 0,
                                                pProperty->m_nVertexCount, 0,
                                                pProperty->m_nIndexCount / 3 );

        } // Next Property Group
    } // Next Light group
}
```

Notice how we pass the pProperty->VertexStart value into the render call to tell the device what value should be added to each index in the property group index buffer before it is used to reference a vertex in the vertex buffer. This makes sure that the indices in the property group (which are zero based and relative to the property group) are mapped to the correct section of the light group vertex buffer.

Study Questions

1. What is a vertex normal and why do we need them?
2. Explain the differences between a spot light, a point light and a directional light. List as many of the characteristics of all three that you can think of.
3. Is it possible for a directional light to attenuate with distance?
4. What is ambient light and is the vertex normal taken into account when reflecting it?
5. Specular highlights depend on the positions and orientation of the viewer. TRUE or FALSE?
6. The D3DLIGHT9 structure has a ***Falloff*** member. This member is only applicable to one light type, which type? (Spot, Point or Directional)
7. Does the DirectX lighting pipeline perform light occlusion to produce shadows?
8. Explain what the attenuation properties of the D3DLIGHT9 structure allow us to control?
9. Do directional lights have a limited range?
10. The emissive color of a material is added to the color of each vertex using that material whether the vertex is within range of a light or not. TRUE or FALSE?
11. What does the ***Power*** member of the D3DMATERIAL9 structure allow us to change the appearance of?
12. When using a Software Vertex Processing Device all lighting calculations are still done on the 3D hardware but just not as fast as with Hardware Vertex Processing. TRUE or FALSE?
13. All Software Vertex Processing Device must use no more than 8 lights. TRUE or FALSE?
14. How can we find out how many simultaneously active lights the hardware supports?
15. Why do you think DirectX lighting (and other similar techniques) are referred too as vertex lighting techniques?
16. If the DirectX lighting pipeline is enabled, why would you wish to store color components at the vertex level, and how could they be used?
17. What is the meant when people comment that you should ‘batch render your primitives’?

Chapter 5 Appendix A Device States

RenderState Type	Argument	Description
D3DRS_LIGHTING	TRUE or FALSE	Enables/Disables the DirectX fixed function lighting pipeline. Vertices passed to DrawPrimitive should have normals that will be used in the lighting calculations. Lights should have been placed into the scene and a material should be set describing the reflectance properties used in the lighting calculation.
D3DRS_AMBIENT	D3DCOLOR	Used to set the color of the global ambient light of the scene. This color is modulated with the ambient reflectance property of the device material (or the vertex) and the result is added to the color of each vertex. This is in addition to any ambient light that may be received by a vertex from nearby light sources.
D3DRS_SPECULARENABLE	TRUE or FALSE	Used to enable or disable specular highlight calculations by the lighting pipeline. Has no effect if lighting is disabled. Specular highlights add view dependant highlights adding to the realism of the scene.

		Specular highlights are more expensive to calculate and so should be disabled when not required.
D3DRS_DIFFUSEMATERIALSOURCE D3DRS_AMBIENTMATERIALSOURCE D3DRS_SPECULARMATERIALSOURCE D3DRS_EMISSIVEMATERIALSOURCE	D3DMATERIALCOLORSOURCE	If lighting is enabled, these four render states tell the lighting pipeline where to get the reflectance properties for the diffuse, ambient, specular, and emissive calculation. They can be set to use the relevant member in the current material, the first color stored in the vertex or the second color stored in the vertex.

Misc Device State Types

Device State Function	Parameters	Description
SetLight	DWORD Index , D3DLIGHT9 * Light	Binds a set of light properties to device light slot [Index].
LightEnable	DWORD Index, BOOL bEnable	Enable/Disable the light at device light slot [Index]. If the lighting pipeline is not enabled, this will have no effect.
SetMaterial	D3DMATERIAL9 *pMaterial	Sets the material properties of the device used by the lighting pipeline. If lighting is disabled then the material has no effect. When lighting is enabled, the material contains the

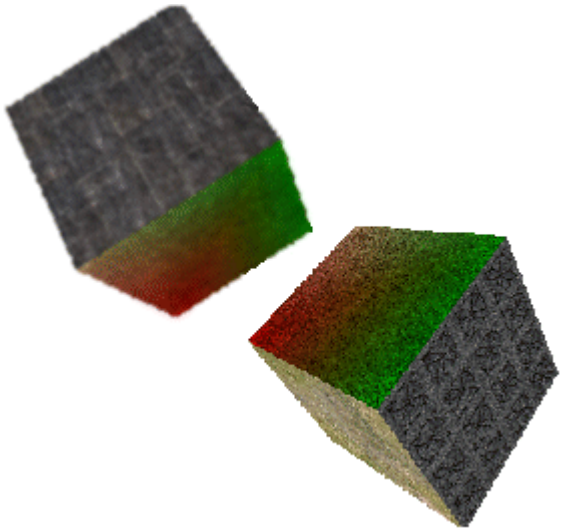
				reflectance properties used by the lighting calculations to determine how much of the incoming light received by a vertex is reflected. This effects the overall color of the vertex.

Workbook Chapter Six:
Texture Mapping

Lab Project 6.1: Basic Texturing

Lab Project 6.1 will demonstrate:

- Loading and creating textures with MIP maps.
- Using vertices with texture coordinates.
- Enabling MIP mapping and setting the minification, magnification, and MIP filters.
- Setting texture states for texture stage 0.
- The construction and rendering of a "Sky Box"



To keep things simple for our first demo, we will render cubes again. The cube on the left will use bilinear filtering for magnification and minification and MIP maps with a linear MIP filter (i.e. trilinear filtering). The cube on the right is rendered without any MIP maps and without any filtering for magnification and minification.

We assigned textures to each face along with green and red colors stored at the vertices. Vertex colors are interpolated across the face and modulated with the texture in the texture stage. The application will use pre-lit vertices with diffuse colors stored at the vertex. Every face but one has all its vertices set to white (0xFFFFFFFF).

The CVertex Class

We add two floats to our CVertex class to store the UV texture coordinates.

```
class CVertex
{
public:
    // Constructors & Destructors for This Class.
    Cvertex (float fX, float fY, float fZ,
             ULONG ulDiffuse = 0xFF000000,
             float ftu = 0.0f, float ftv = 0.0f )
             { x = fX; y = fY; z = fZ; Diffuse = ulDiffuse;
               tu = ftu; tv = ftv; }

    CVertex(){x = 0.0f; y = 0.0f; z = 0.0f;
              Diffuse = 0xFF000000; tu = 0.0f; tv = 0.0f; }

    // Public Variables for This Class
    float       x;                  // Vertex Position X Component
```

```
    float       y;                  // Vertex Position Y Component
    float       z;                  // Vertex Position Z Component
    ULONG       Diffuse;            // Diffuse colour component
    float       tu;                 // Texture u coordinate
    float       tv;                 // Texture v coordinate
  };
```

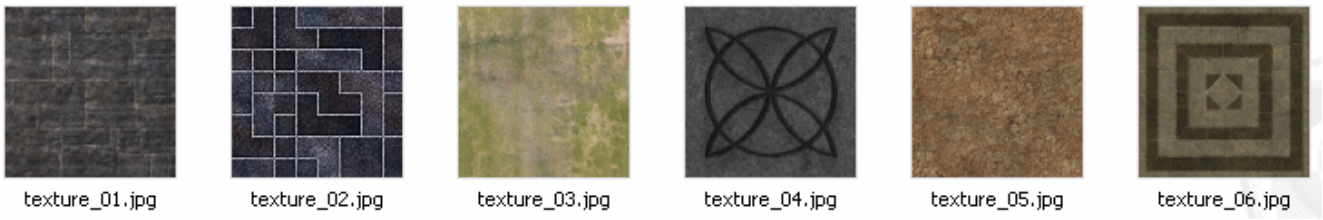
The constructors have been altered to allow input texture coordinates for each vertex created. We are not using the lighting pipeline, so we store a diffuse color at the vertex. The vertex flags are defined in CObjects.h:

```
#define VERTEX_FVF   D3DFVF_XYZ | D3DFVF_DIFFUSE | D3DFVF_TEX1
```

We are using the **D3DFVF_TEX1** flag to inform the device that we have one set of texture coordinates in the vertex.

The CGameApp Class

The CGameApp class now includes an array of IDirect3DTexture9 interfaces. This will hold all of the textures that our demo will use. The textures themselves are in the /Data subdirectory and are listed below along with their filenames.




```
LPDIRECT3DTEXTURE9      m_pTextures[6];   // Store six texture pointers here
LPDIRECT3DVERTEXBUFFER9 m_pVertexBuffer;  // Vertex Buffer to be Rendered
CObject                 m_pObject[2];     // Objects storing mesh instances
```

CGameApp::BuildObjects

In this demo we will duplicate vertices for each face rather than use the indexed strip method from Chapter 3. We do this because we wish to use a different texture for each face and require unique texture coordinates. The first part of the function is unchanged. We create a vertex buffer large enough to hold all the vertices and then create and lock it as shown below.

```
bool CGameApp::BuildObjects()
{
    HRESULT  hRet;
    CVertex *pVertex = NULL;
    ULONG    ulUsage = D3DUSAGE_WRITEONLY;
```

```
    // Seed the random number generator
    srand( timeGetTime() );

    // Release previously built objects
    ReleaseObjects();

    // Build our buffers usage flags (i.e. Software T&L etc)
    VERTEXPROCESSING_TYPE vp;
    vp = m_D3DSettings.GetSettings()->VertexProcessingType;

    if ( vp != HARDWARE_VP && vp != PURE_HARDWARE_VP )
         ulUsage |= D3DUSAGE_SOFTWAREPROCESSING;

    // Create our vertex buffer ( 24 vertices (4 verts * 6 faces) )
    hRet = m_pD3DDevice->CreateVertexBuffer(sizeof(CVertex) * 24,
                                            ulUsage, VERTEX_FVF,
                                            D3DPOOL_MANAGED,
                                            &m_pVertexBuffer, NULL );
    if ( FAILED( hRet ) ) return false;

    // Lock the vertex buffer ready to fill data
    hRet = m_pVertexBuffer->Lock(0, sizeof(CVertex)*24,(void**)&pVertex,0);
    if ( FAILED( hRet ) ) return false;
```

Below we see the first three faces created. Notice the vertex winding order. We are using one strip per-face. The front, back and top faces will have their vertex colors set to white. The textures that these faces use will thus be unaltered by the modulation in the texture stage and the texture colors will be copied to these surfaces unmodified. Each face has their texture coordinates set to map to the four corners of the texture and thus the entire texture will be mapped to the surface with no tiling.

```
    // Front Face
    *pVertex++ = CVertex( -2, -2, -2, 0xFFFFFFFF, 0.0f, 1.0f );
    *pVertex++ = CVertex( -2,  2, -2, 0xFFFFFFFF, 0.0f, 0.0f );
    *pVertex++ = CVertex(  2, -2, -2, 0xFFFFFFFF, 1.0f, 1.0f );
    *pVertex++ = CVertex(  2,  2, -2, 0xFFFFFFFF, 1.0f, 0.0f );

    // Top Face
    *pVertex++ = CVertex( -2,  2, -2, 0xFFFFFFFF, 0.0f, 1.0f );
    *pVertex++ = CVertex( -2,  2,  2, 0xFFFFFFFF, 0.0f, 0.0f );
    *pVertex++ = CVertex(  2,  2, -2, 0xFFFFFFFF, 1.0f, 1.0f );
    *pVertex++ = CVertex(  2,  2,  2, 0xFFFFFFFF, 1.0f, 0.0f );

    // Back Face
    *pVertex++ = CVertex( -2,  2,  2, 0xFFFFFFFF, 0.0f, 1.0f );
    *pVertex++ = CVertex( -2, -2,  2, 0xFFFFFFFF, 0.0f, 0.0f );
    *pVertex++ = CVertex(  2,  2,  2, 0xFFFFFFFF, 1.0f, 1.0f );
    *pVertex++ = CVertex(  2, -2,  2, 0xFFFFFFFF, 1.0f, 0.0f );
```

The bottom face uses a white diffuse color also but the texture coordinates are in the range [0.0, 4.0] along the U and V axes. This means that the texture will be tiled 4 times horizontally and vertically across the face.

```
    // Bottom Face
    *pVertex++ = CVertex( -2, -2,  2, 0xFFFFFFFF, 0.0f, 4.0f );
```

```
    *pVertex++ = CVertex( -2, -2, -2, 0xFFFFFFFF, 0.0f, 0.0f );
    *pVertex++ = CVertex(  2, -2,  2, 0xFFFFFFFF, 4.0f, 4.0f );
    *pVertex++ = CVertex(  2, -2, -2, 0xFFFFFFFF, 4.0f, 0.0f );
```

The left face uses diffuse vertex colors so that you can see the color blending taking place in the texture stage. We assign a pure green diffuse color to the bottom two vertices in the face, a red color to the top right face, and a white diffuse color to the top left vertex.

```
    // Left Face
    *pVertex++ = CVertex( -2, -2,  2, 0xFF00FF00, 0.0f, 1.0f );
    *pVertex++ = CVertex( -2,  2,  2, 0xFFFF0000, 0.0f, 0.0f );
    *pVertex++ = CVertex( -2, -2, -2, 0xFF00FF00, 1.0f, 1.0f );
    *pVertex++ = CVertex( -2,  2, -2, 0xFFFFFFFF, 1.0f, 0.0f );
```

Finally, the right face has all white vertices but they are not mapped to the four corners of the texture. Instead they are mapped to the square in the middle of the brown texture with the diamond shape on it. Because the texture coordinates are in the range [0.4, 0.6] this takes the middle region of the texture and stretches it to fill the entire face. This face is a good test of the magnification filters. The right cube is not using bilinear filtering and the diamond edges looks blocky when you move the camera up close to it.

```
    // Right Face
    *pVertex++ = CVertex(  2, -2, -2, 0xFFFFFFFF, 0.4f, 0.6f );
    *pVertex++ = CVertex(  2,  2, -2, 0xFFFFFFFF, 0.4f, 0.4f );
    *pVertex++ = CVertex(  2, -2,  2, 0xFFFFFFFF, 0.6f, 0.6f );
    *pVertex++ = CVertex(  2,  2,  2, 0xFFFFFFFF, 0.6f, 0.4f );

    // Unlock the buffer
    m_pVertexBuffer->Unlock( );

    // Our two objects should reference this vertex buffer
    m_pObject[ 0 ].SetVertexBuffer( m_pVertexBuffer );
    m_pObject[ 1 ].SetVertexBuffer( m_pVertexBuffer );
```

We assigned the same vertex buffer to both objects and now we set the world matrix for each object to their initial positions and orientations.

```
    // Set both objects matrices so that they are offset slightly
    D3DXMatrixTranslation(&m_pObject[ 0 ].m_mtxWorld, -2.5f,  2.0f, 10.0f );
    D3DXMatrixTranslation(&m_pObject[ 1 ].m_mtxWorld,  2.5f, -2.0f, 10.0f );
```

Finally, we load all the textures this application uses and store them in the CGameApp's IDirect3DTexture9 array.

```
    // Load all 6 textures used in this example.
    hRet = D3DXCreateTextureFromFile( m_pD3DDevice, "Data\\texture_01.jpg",
                                      &m_pTextures[0] );
    if ( FAILED(hRet) ) return false;
    hRet = D3DXCreateTextureFromFile( m_pD3DDevice, "Data\\texture_02.jpg",
                                      &m_pTextures[1] );
```

```
    if ( FAILED(hRet) ) return false;
    hRet = D3DXCreateTextureFromFile( m_pD3DDevice, "Data\\texture_03.jpg",
                                      &m_pTextures[2] );
    if ( FAILED(hRet) ) return false;
    hRet = D3DXCreateTextureFromFile( m_pD3DDevice, "Data\\texture_04.jpg",
                                      &m_pTextures[3] );
    if ( FAILED(hRet) ) return false;
    hRet = D3DXCreateTextureFromFile( m_pD3DDevice, "Data\\texture_05.jpg",
                                      &m_pTextures[4] );
    if ( FAILED(hRet) ) return false;
    hRet = D3DXCreateTextureFromFile( m_pD3DDevice, "Data\\texture_06.jpg",
                                      &m_pTextures[5] );
    if ( FAILED(hRet) ) return false;

    // Success!
    return true;
}
```

We used the simplified version of the D3DXCreateTextureFromFile function to load the textures. This function will generate the MIP chain for each texture and will store them in the **D3DPOOL_MANAGED** pool. This is what we want because we will not have to bother reloading them if the device becomes lost.

CGameApp::SetupRenderStates

CGameApp:SetupGameStates now includes additional code to setup the texture stage states.

```
void CGameApp::SetupRenderStates()
{
    HRESULT  hRet;
    D3DCAPS9 Caps;

    // Set up new perspective projection matrix
    float fAspect = (float)m_nViewWidth / (float)m_nViewHeight;
    D3DXMatrixPerspectiveFovLH(&m_mtxProjection, D3DXToRadian( 60.0f ),
                               fAspect, 1.01f, 1000.0f );

    // Setup our D3D Device initial states
    m_pD3DDevice->SetRenderState( D3DRS_ZENABLE, D3DZB_TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_DITHERENABLE,  TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_SHADEMODE, D3DSHADE_GOURAUD );
    m_pD3DDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CCW );
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, FALSE );
```

Although these next states are the default states for stage 0, we explicitly set them anyway just in case the driver is not as well behaved as it should be (which has been known to happen from time to time).

```
    // Setup our Texture Stage States
    m_pD3DDevice->SetTextureStageState(0,D3DTSS_COLOROP,D3DTOP_MODULATE);
    m_pD3DDevice->SetTextureStageState(0,D3DTSS_COLORARG1,D3DTA_DIFFUSE);
    m_pD3DDevice->SetTextureStageState(0,D3DTSS_COLORARG2,D3DTA_TEXTURE);
```

We let the device know about our new flexible vertex format with the UV coordinate.

```
    // Setup our vertex FVF code
    m_pD3DDevice->SetFVF( VERTEX_FVF );

    // Setup our matrices
    m_pD3DDevice->SetTransform( D3DTS_VIEW, &m_mtxView );
    m_pD3DDevice->SetTransform( D3DTS_PROJECTION, &m_mtxProjection );
```

Although we would normally enable our MIP map, magnification, and minification filters in this function, we deliberately have not done so in this demo. We will enable them when we render the left cube and disable them for the right cube so that we can see the differences. What we will do however is query the device caps to make sure that the magnification, minification, and MIP filters support the D3DTF_LINEAR capabilities. This way we will know if they can be safely enabled for the left cube. We did not put this test in CMyD3DInit::ValidateDevice because we would not want to reject the device from enumeration and possible selection if these filters were not supported (which will probably almost never be the case since even most older 3D cards included bilinear filtering and MIP map support).

```
    // Retrieve device caps, the following states are not required
    hret = m_pD3D->GetDeviceCaps(
                               m_D3DSettings.GetSettings()->AdapterOrdinal,
                               m_D3DSettings.GetSettings()->DeviceType,
                               &Caps);
    if ( FAILED(hRet) ) return;

    // Determine if linear filtering is supported
    m_bFilterEnabled = false;

    if((Caps.TextureFilterCaps & D3DPTFILTERCAPS_MAGFLINEAR) &&
       (Caps.TextureFilterCaps & D3DPTFILTERCAPS_MINFLINEAR))
         m_bFilterEnabled = true;

    // Determine if linear mip-mapping is supported
    m_bMipEnabled = false;
    if(Caps.TextureFilterCaps & D3DPTFILTERCAPS_MIPFLINEAR)
                               m_bMipEnabled = true;
}
```

CGameApp::FrameAdvance

The first part of the function is unchanged from the previous cube demos. We call ProcessInput to update the view matrix if the user has pressed the cursor keys. Then we call the AnimateObjects function to build each world matrix. Finally, we clear the frame buffer and prepare to render our new scene.

```
    // Poll & Process input devices
    ProcessInput();
```

```
    // Animate the two objects
    AnimateObjects();

    // Clear the frame & depth buffer ready for drawing
    m_pD3DDevice->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                        0xFFFFFFFF, 1.0f, 0 );
```

We are about to render the first cube, so we will enable the supported filters. This is why we recorded the boolean variable during the SetupRenderStates function.

```
    // Begin Scene Rendering
    m_pD3DDevice->BeginScene();

    // Enable Linear Filter for object[0] if supported
    if ( m_bFilterEnabled )
    {
        m_pD3DDevice->SetSamplerState(0,D3DSAMP_MAGFILTER,D3DTEXF_LINEAR);
        m_pD3DDevice->SetSamplerState(0,D3DSAMP_MINFILTER,D3DTEXF_LINEAR);
    }

    // Enable mip-mapping for object[0] if supported
    if ( m_bMipEnabled )
    {
        m_pD3DDevice->SetSamplerState(0,D3DSAMP_MIPFILTER,D3DTEXF_LINEAR);
    }
```

Now we loop through each object and set its world matrix and vertex buffer.

```
    // Loop through each object
    for ( ULONG i = 0; i < 2; i++ )
    {
        // Set our object matrix
        m_pD3DDevice->SetTransform(D3DTS_WORLD, &m_pObject[i].m_mtxWorld);

        // Set the vertex stream source
        m_pD3DDevice->SetStreamSource(0, m_pObject[i].m_pVertexBuffer, 0,
                                      sizeof(CVertex) );
```

For each object we loop through each of its six faces. For each face, we assign the texture to texture stage 0 by calling SetTexture and then render the face as a triangle strip.

```
        // Render one face per texture
        for ( ULONG j = 0; j < 6; j++ )
        {
            // Set the texture for this primitive
            m_pD3DDevice->SetTexture( 0, m_pTextures[j] );

            // Render the primitive (1 strip per face in the vertex buffer)
            m_pD3DDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, j * 4, 2 );
        }
```

With the cube rendered, we now disable filtering so that when the next cube is drawn, MIP maps are disabled and the minification and magnification filters are set to point sampling. When the loop exits, we present the scene.

```
        // Disable linear filtering and mip-maps (for object[1])
        m_pD3DDevice->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_POINT);
        m_pD3DDevice->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_POINT);
        m_pD3DDevice->SetSamplerState(0, D3DSAMP_MIPFILTER, D3DTEXF_NONE);

    } // Next Object

    // End Scene Rendering
    m_pD3DDevice->EndScene();

    // Present the buffer
    if(FAILED(m_pD3DDevice->Present( NULL, NULL, NULL, NULL )))
        m_bLostDevice = true;
}
```

Before continuing with the next project, experiment with the code. Try out some of the other texture blending operations. Set your own vertex colors and try new texture coordinates for each face to see the results.

Lab Project 6.2: Multi-Texturing

In Lab Project 3.2 we created a terrain using a height map. We used the values stored at each pixel in the height map to set the height of the corresponding pixel in the mesh. The terrain was not textured but it did include a color at each vertex. This was determined using simple lighting equations when the terrain was constructed. In Lab Project 5.3 we replaced the pre-calculated vertex lighting code and began using DirectX lighting. In this project we will apply two textures to the terrain. Note that we have removed the lighting from the demo so that it does not distract us from our studies. You should have no trouble adjusting the code to re-enable lighting (when you do so in your exercises).

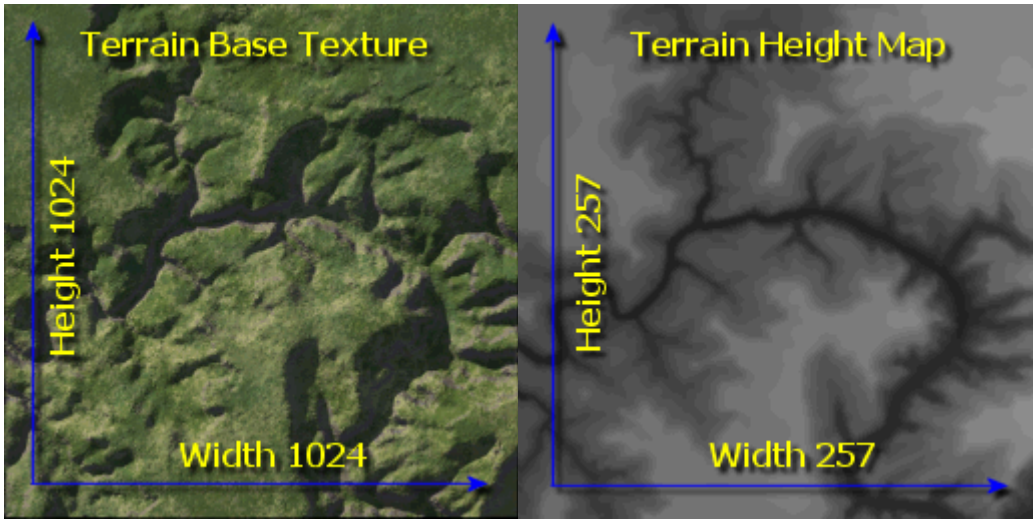
The first thing we need to do is to generate a terrain texture. This is not as hard as you might think. There is a wonderful terrain generation package called Terragen™ which will make our job very easy. You can download a shareware version for free from the following website:

`http://www.planetside.co.uk/terragen/`

This is the package we used to generate the texture for this application. Terragen™ not only enables you to build textures for your terrain, but it also allows you to create a height map for your terrain as well. Appendix A walks you quickly through the process of using Terragen™ to generate terrain textures and height maps if you are unfamiliar with the application.

Texturing a Terrain

In the following images we see a terrain texture that was created using Terragen™ (left). The image on the right reminds us what our height map looks like. This is the same height map used in previous lab projects. Notice how Terragen has created a texture that is synchronized with our height map.



The height map is 257x257 pixels. Each pixel represents a vertex in our terrain. The texture that will be mapped to the terrain however is much bigger (1024x1024). We use a large texture because it will be stretched over the entire terrain. Given the large area of our terrain, a larger texture will provide for more surface detail. You might recall that in flight simulation games not long ago, the terrain looked either too blurry or too blocky when you flew low to the ground. This was because the textures were being stretched over several miles of terrain and texels were mapped to large areas of land. While we will use filters to minimize blockiness in our project, the terrain will look quite blurry due to the size of a single texel in the game world. We will discuss how to address this concern shortly.

Generating Terrain Texture Coordinates

Applying the base texture to the terrain is remarkably easy regardless of the size of the texture as we will essentially drape it over the entire terrain. The four corners of the texture will be mapped to the four corners of the terrain patch. Our vertex structure will need to hold a pair of texture coordinates for the base texture and we will look at our revised vertex structure in a moment. We know that the top left corner of the texture is UV coordinate (0, 0) and that the bottom right corner of the texture is UV coordinate (1, 1). Therefore, every other vertex in between will have UV coordinates within that range.

In order to calculate the U texture coordinate of each vertex, we will take the index of the vertex in the current row of the terrain and divide this by the number of vertices in a row (257 in this case). For example, we know that the top right vertex of the terrain should be mapped to the top right corner of the texture. This is the 257th vertex in the row

```
U = VertexRowOffset / NumberOfVerticesInARow
U = 257 / 257;
U = 1.0;          // top right texel of texture
```

If this is the first vertex in the row then the U coordinate should be 0:

```
U= VertexRowOffset / NumberOfVerticesInARow
U= 0 / 257
U= 0.0 // top left texel in the texture
```

If the vertex was halfway through a row (row position 128 approximately), then the U coordinate should be roughly halfway through the texture (~0.5).

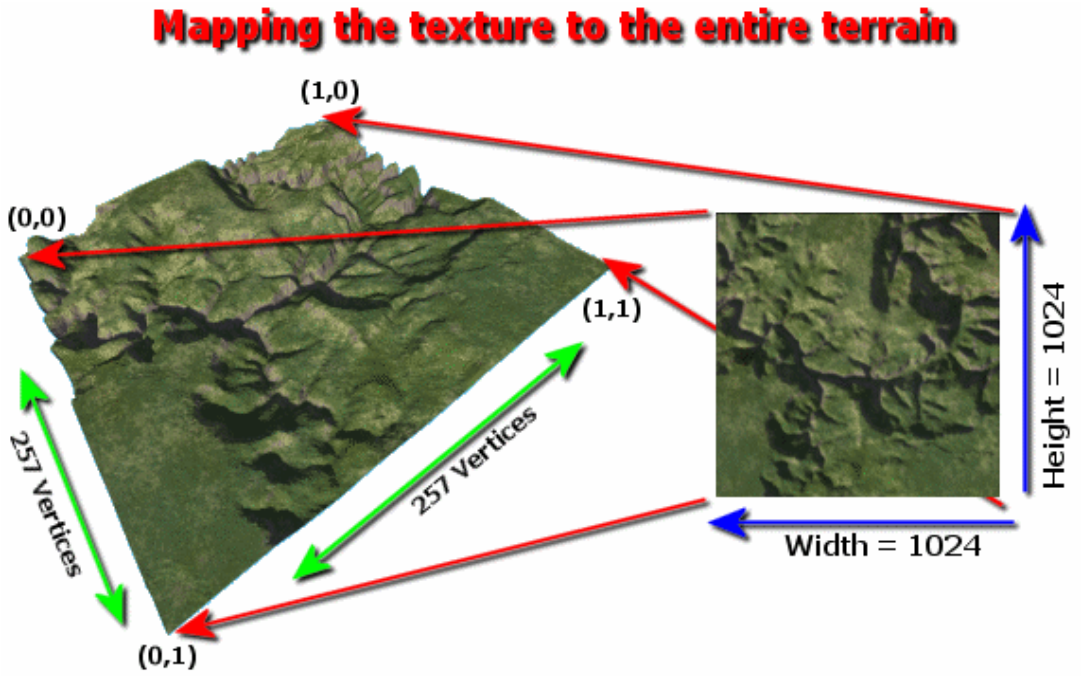
```
U = VertexRowOffset / NumberOfVerticesInRow
U = 128 / 257
U = 0.498  // approximately halfway across the texture horizontally
```

The process works identically for the V coordinate. We divide the position of the vertex vertically by the number of rows in our terrain to get the V coordinate. If we had a 257x129 terrain of vertices, we could calculate the UV coordinates for each vertex as follows.

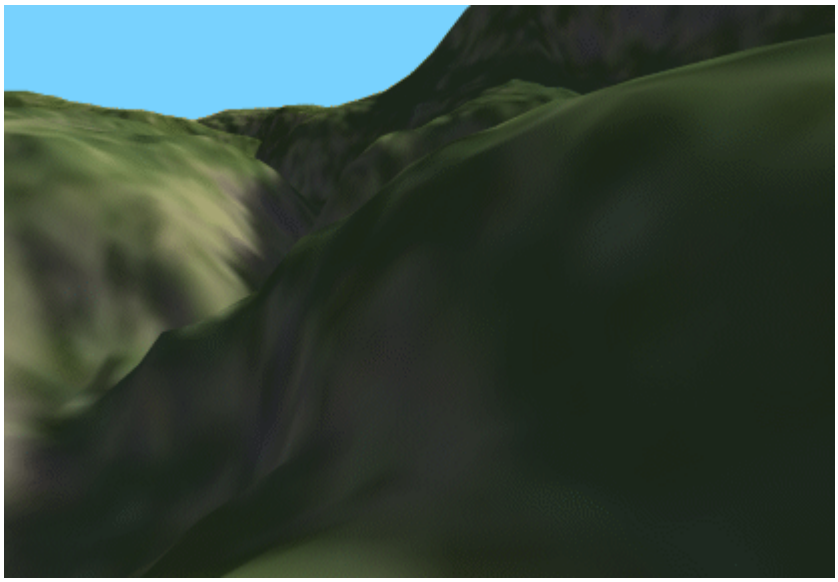
```
for ( Z = 0 ; Z < 129 ; Z++)  // For Each Row
{
      for (X = 0 ; X < 257 ; X++)   // For Each Column
      {
            Vertex[Z][X].u = X / 257;
            Vertex[Z][X].v = Z / 129;
      }
}
```

> **Note:** To keep things simple for now, the code assumes that the vertices are stored in a multi-dimensional array in the form: Vertex[Row][Column]. In reality they will be stored in a vertex buffer.

This loop will generate a UV coordinate for every vertex in the terrain. The range for each UV coordinate would be [0.0, 1.0]. Using our 257x257 terrain, the texture would be mapped to the terrain as shown in the following image:

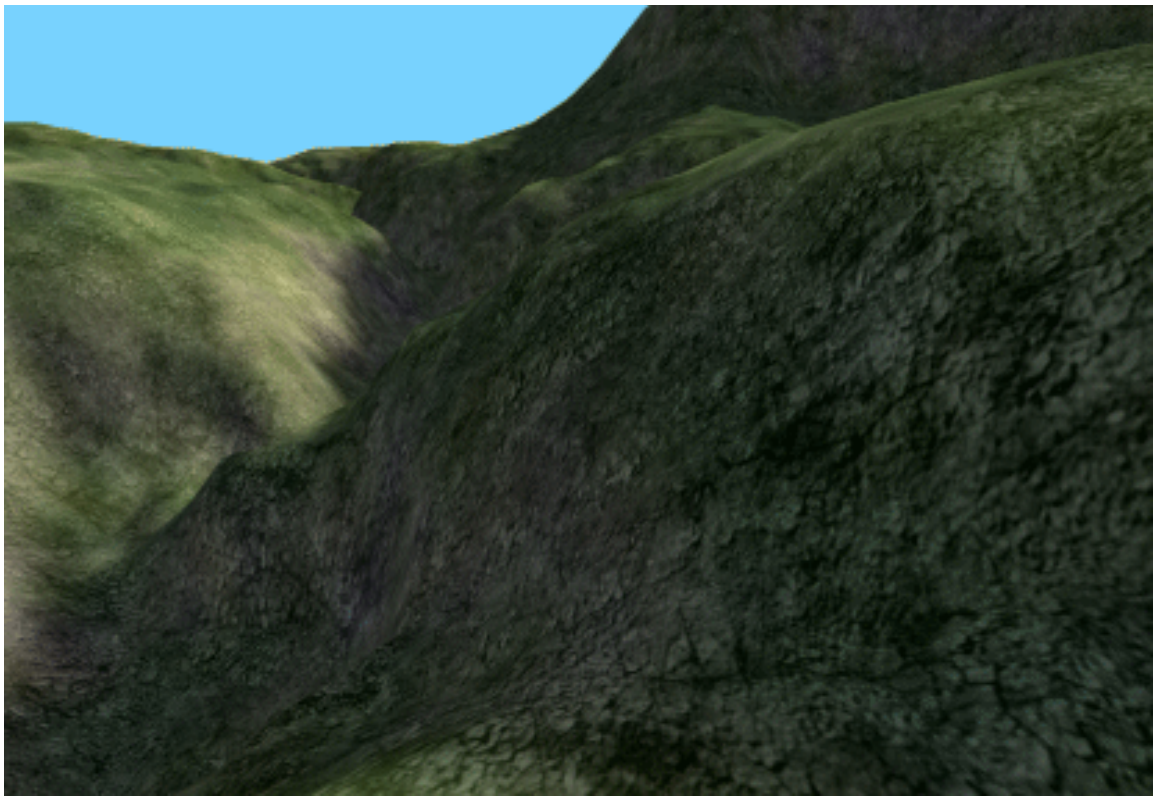


While this terrain texture is fairly large, it still is not big enough to provide enough detail when viewed up close. The solution is not to simply build a larger texture because many cards do not support textures larger than 1024x1024 (some cards do not even support textures larger than 256x256). But even if we could store a large texture that provided enough detail (4000x4000) for example, the memory costs would be considerable.



As it stands now, our 1024x1024 texture will look blurred when viewed up close as this image demonstrates.

While the image above is not necessarily a terrible sight, we can certainly do much better…



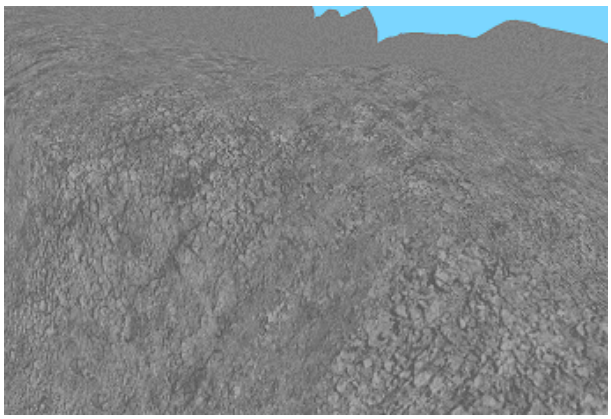
This second image looks a good deal better. The solution was a simply multi-texturing technique called **detail mapping**.

Single Pass Detail Mapping

A detail map is a tileable texture with a high-frequency pattern. This project uses a 512x512 detail texture map which is shown below.

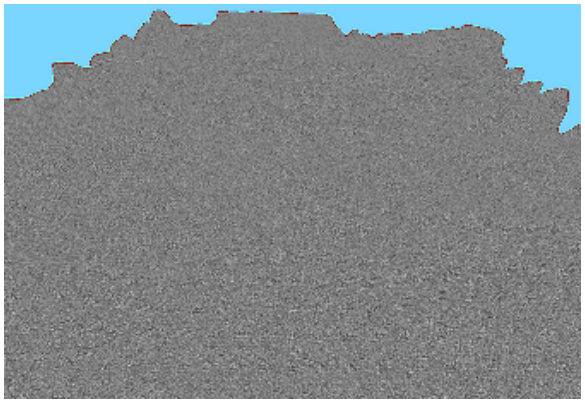
This is a texture that will tile quite nicely. We certainly cannot simply stretch this texture over the terrain as we did our base texture or it would suffer from the same problem. In fact it would look worse because this texture is much smaller. However you should recall from the text that when we specify texture coordinates outside the 0.0 to 1.0 range with wrap texture addressing mode enabled, the texture will repeat across the polygons. So we can calculate a second set of texture coordinates for our vertices that would tile the detail texture across the terrain many times over. This means that the detail texture would not be stretched. Even when viewed up close, a sizeable section of the detail texture is mapped to our immediate vicinity as shown below. The image to the right shows only the detail texture applied to the terrain.

As you can see, even when standing right next to the terrain there is still plenty of texture to see. When blended with our terrain texture, this detail map will provide a much more visually compelling terrain.



In the next image you can see what this look like when we zoom out a bit. With MIP mapping disabled we are able to see quite clearly the amount of detail we will be adding.

The detail map selected for this project is somewhat terrain specific. You will note what look like small rocks and similar pieces of debris that make sense for the effect we want to achieve. However, detail maps are certainly not limited to terrain rendering. They can be used in any scene to achieve the same purpose. For non-terrain scenes you will probably prefer more of a random noise pattern. You will find many free resources on the Internet that would serve as good detail maps or you can just as easily generate one in a paint program. A gray cement/concrete texture that tiles without seams would make a pretty good detail map for scenes with brick walls for example.



The image to the left demonstrates that even one tile of detail texture significantly changes the way we perceive the terrain underneath it. Unlike the base texture which has limited texels to devote to this terrain section, the detail map is mapped in its entirety to the section highlighted in the white box.

The image on the right depicts how the detail map will be tiled over the terrain by using texture coordinates that tile. The borders are drawn in only to make it easier to see the effect; they would certainly not be there in the actual application. In this project we will tile the detail texture approximately 43 times horizontally and the same vertically. As a result, every 5 quad square of the terrain will have its own full detail map.

Let us now begin examining the code for this project.

The CVertex Class

Because we are not using lighting or vertex colors in this first project, we only need our vertex to hold an XYZ position and two sets of 2D texture coordinates. The vertex class definition is shown below and can be found in CObjects.h

```
class CVertex
{
public:
    // Constructors & Destructors for This Class.
    CVertex(float fX, float fY, float fZ, float ftu=0.0f, float ftv=0.0f)
        { x = fX; y = fY; z = fZ; tu = ftu; tv = ftv; }

    CVertex() { x = 0.0f; y = 0.0f; z = 0.0f; tu = 0.0f; tv = 0.0f; }

    // Public Variables for This Class
    float       x;          // Vertex Position X Component
    float       y;          // Vertex Position Y Component
    float       z;          // Vertex Position Z Component
    float       tu;         // Texture u coordinate
    float       tv;         // Texture v coordinate
    float       tu2;        // 2nd Texture U coordinate
    float       tv2;        // 2nd Texture V coordinate
};.
```

The flexible vertex flags to accompany this vertex structure will need to indicate that our vertices now have two sets of 2D vertex coordinates:

```
#define VERTEX_FVF  D3DFVF_XYZ | D3DFVF_TEX2
```

Note: Although the terrain is storing texture coordinates, we will still use the old pre-lit vertex structure to render our CPlayer object (the cube). This means we will need to change vertex types with a call to IDirect3DDevice9::SetFVF between rendering the terrain and the CPlayer object. If you look in the CObject.h file you will see that we also have a set of FVF flags to describe the pre-lit vertex format the CPlayer mesh uses.

```
#define LITVERTEX_FVF  D3DFVF_XYZ | D3DFVF_DIFFUSE

class CLitVertex
{
public:
   // Constructors & Destructors for This Class.
   CLitVertex( float fX, float fY, float fZ, ULONG ulDiffuse = 0xFF000000 )
                    { x = fX; y = fY; z = fZ; Diffuse = ulDiffuse; }

   CLitVertex() { x = 0.0f; y = 0.0f; z = 0.0f; Diffuse = 0xFF000000; }

   // Public Variables for This Class
   float      x, y, z;
   ULONG      Diffuse;
};
```

The base texture will be set in texture stage 0. Stage 0 will by default use the first set of texture coordinates in the vertex to map the base texture to the terrain. In the second stage, we will assign the detail map. This stage by default will use the second set of texture coordinates. The first set of texture coordinates will all be in the [0.0, 1.0] range. The second set of texture coordinates will use a larger range to tile the texture. The texture stage setup will look like the following:

```
// Set texture addressing and color ops
m_pD3DDevice->SetTexture ( 0 , pBaseTexture);
m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
m_pD3DDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );

m_pD3DDevice->SetTexture(  , pDetailTexture);
m_pD3DDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_CURRENT );
m_pD3DDevice->SetTextureStageState( 1, D3DTSS_COLORARG2, D3DTA_TEXTURE );
m_pD3DDevice->SetTextureStageState( 1, D3DTSS_COLOROP, D3DTOP_ADDSIGNED );
m_pD3DDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX, 1 );
```

In the above code we bind the base texture to stage 0 as **ARG1**. We then use the **D3DTOP_SELECTARG1** color operation to take the sampled texture color and output it to the next stage. If we were using DirectX lighting (or vertex colors) you would want assign **D3DTA_DIFFUSE** to **ARG2** and modulate the texture and diffuse color before sending the result on to the next stage. Finally, although it is the default state, we inform the device that the first set of texture coordinates in the vertex are the ones that should be used to sample the texture in this stage.

We assigned the detail texture to stage 1 as input argument **ARG2**. We use the add-signed color operation because it is ideal for detail mapping. If this operation is not supported by the current hardware we could use a modulate2X instead to achieve similar results.

Provided the hardware supports at least two simultaneous textures, this is all we will need to do. We do not need to enable alpha blending or do any frame buffer blending because all of the color blending is done in the texture stages. If the system does not support multi-texturing then we will need to render the terrain in two passes (because texture stage 1 will not be available to us).

Multi-Pass Detail Mapping

If the device does not support multi-texturing then we will be limited to using just texture stage 0. We will have to render our polygons using a single texture at a time. When this is the case our application will have to do the following in our render loop.

1. Assign base texture map to stage 0
2. Set texture stage 0 to use the first set of texture coordinates in our vertex
3. Set the texture stage color operation to use the texture color assigned to that stage
4. Render the terrain
5. Assign detail texture map to stage 0
6. Set texture stage 0 to use second set of texture coordinates in the vertex
7. Set texture stage color operation to use the texture color
8. Enable alpha blending
9. Set the source and destination blend modes for frame buffer blending
10. Render the terrain again
11. Disable alpha blending
12. Continue with render loop

This form of multi-texturing is referred to as multiple pass texture blending (or simply multi-pass blending). Performance will suffer some because we have to render the terrain twice -- once with the base texture and once with the detail map. When we render it the second time, we enable alpha blending to blend the polygons rendered in the second pass with the polygons in the frame buffer that exist from the previous render pass.

In the text we discussed the fact that setting the source blend factor to the frame buffer color and the destination blend factor to the source color, we generate a modulate2X blending function. This will provide similar results to the add-signed blending used in the single pass case. A multiple pass technique might look similar to the code listed below.

```
m_pD3DDevice->SetTexture( 0 , pBaseTexture );
m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLOROP  , D3DTOP_SELECTARG1 );
m_pD3DDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );
RenderTerrain();
```

We set the base texture in stage 0 and setup the color operation to output the color. We also set the stage to use the first set of texture coordinates. Then we render the first pass of the terrain.

Now we need to render the terrain one more time. So we will set the detail texture and make sure that the second set of texture coordinates are used to map the detail texture to the terrain. Once again we simply output the color from the stage:

```
m_pD3DDevice->SetTexture( 0 , pDetialTexture );
m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
```

```
m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
m_pD3DDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 1 );
```

Finally, we enable alpha blending, setup our frame buffer blend modes and re-render the terrain.

```
m_pD3DDevice->SetRenderState( D3DRS_SRCBLEND ,  D3DBLEND_ONE );
m_pD3DDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_ZERO );
m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );

RenderTerrain();

m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );
```

The CMesh Class

Since we will be using two different types of vertices in our application (one for the terrain vertices and one for the CPlayer mesh) the CMesh class will have to be modified slightly so that it can handle arbitrary vertex formats.

```
class CMesh
{
public:
    // Constructors & Destructors for This Class.
    CMesh( ULONG VertexCount, ULONG IndexCount );
    CMesh();
                              virtual ~CMesh();

    // Public Functions for This Class
    void        SetVertexFormat ( ULONG FVFCode, UCHAR Stride );
    long        AddVertex       ( ULONG Count = 1 );
    long        AddIndex        ( ULONG Count = 1 );
    HRESULT     BuildBuffers(LPDIRECT3DDEVICE9 pD3DDevice, bool HardwareTnL,
                             bool ReleaseOriginals = true );

    // Public Variables for This Class
    ULONG                    m_nVertexCount;  // Number of vertices stored
    UCHAR                   *m_pVertex;       // Temporary vertex array of any format
    ULONG                    m_nIndexCount;   //  Number of indices stored
    USHORT                  *m_pIndex;        //  Simple temporary index array
    LPDIRECT3DVERTEXBUFFER9 m_pVertexBuffer; // Vertex Buffer to be Rendered
    LPDIRECT3DINDEXBUFFER9  m_pIndexBuffer;  // Index Buffer to be Rendered
    UCHAR                    m_nStride;       // The stride of each vertex
    ULONG                    m_nFVFCode;      // Flexible vertex format code
};
```

Recall that the CMesh class uses a temporary CVertex array to store vertices until the BuildBuffers function is called to create a vertex buffer. The modified class now points to a vertex array using a UCHAR pointer. We also store the **stride** (the size of the vertex structure) and the FVF flags that describe the vertices in the mesh. Now the BuildBuffers function will know how to create the vertex buffer regardless of vertex type. When filling the temporary array, we can simply cast the UCHAR pointer to a pointer of the type of vertex we intend to store and then write to the array as usual. Also

note that the AddVertex function has been modified. In previous projects it only knew how to allocate enough space for a CVertex structure. Now it will use the stride to determine how many bytes to allocate per vertex. Therefore it is important that you set the stride before adding vertices.

The CGameApp Class

Below you will see that we have grayed out the functions and members that have been in the class for a while and that you should be familiar with at this point. We show only the new functions and member variables added to this demo.

```
class CGameApp
{
public:
    // Constructors & Destructors for This Class.
       CGameApp();
       virtual ~CGameApp();

    // Public Functions for This Class
    LRESULT DisplayWndProc( HWND hWnd, UINT Message, WPARAM wParam, LPARAM lParam );
    bool    InitInstance( HANDLE hInstance, LPCTSTR lpCmdLine, int iCmdShow );
    int     BeginGame( );
    bool    ShutDown( );

private:
    // Private Functions for This Class
    bool        BuildObjects      ( );
    void        ReleaseObjects    ( );
    void        FrameAdvance      ( );
    bool        CreateDisplay     ( );
    void        ChangeDevice      ( );
    void        SetupGameState    ( );
    void        SetupRenderStates ( );
    void        AnimateObjects    ( );
    void        ProcessInput      ( );
    bool        TestDeviceCaps( );  // Test which filters, blend modes are supported
    void        SelectMenuItems( ); // Setup initial menu options

    // Private Static Functions For This Class
    static LRESULT CALLBACK StaticWndProc(HWND hWnd, UINT Message, WPARAM wParam,
                                          LPARAM lParam);

    // Private Variables For This Class
    CTerrain   m_Terrain;  // Simple terrain object (stores data)
    CPlayer    m_Player;   // Player class used to manipulate our player object
    CCamera   *m_pCamera;  // A cached copy of the camera attached to the player

    D3DXMATRIX  m_mtxIdentity;      // A basic identity matrix
    CTimer      m_Timer;            // Game timer
    ULONG       m_LastFrameRate;    // Used for making sure we update when fps changes
    HWND        m_hWnd;             // Main window HWND
    HICON       m_hIcon;            // Window Icon
    HMENU       m_hMenu;            // Window Menu
    bool        m_bLostDevice;      // Is device currently lost ?
    bool        m_bActive;          // Is the application active ?
```

```
    LPDIRECT3D9           m_pD3D;            // Direct3D Object
    LPDIRECT3DDEVICE9     m_pD3DDevice;      // Direct3D Device Object
    CD3DSettings          m_D3DSettings;     // Settings used to initialize D3D

    D3DFILLMODE           m_FillMode;        // fill mode we are using
    D3DTEXTUREOP          m_ColorOp;         // color op we are using
    D3DTEXTUREFILTERTYPE  m_MagFilter;       // Magnification Filter to use
    D3DTEXTUREFILTERTYPE  m_MinFilter;       // Minification Filter to use
    D3DTEXTUREFILTERTYPE  m_MipFilter;       // Mip-Map filter to use
    ULONG                 m_Anisotropy;      // Anisotropy level to use
    bool                  m_bSinglePass;     // Use single pass rendering

    bool m_MagFilterCaps[10]; // Capabilities supported for required filters
    bool m_MinFilterCaps[10]; // Capabilities supported for required filters
    bool m_MipFilterCaps[10]; // Capabilities supported for required mip filters
    bool m_ColorOpCaps[30];   // Capabilities supported for required color ops
    ULONG m_MaxTextures;      // Capabilities supported for required pass count
    ULONG m_MaxAnisotropy;    // Capabilities supported for anisotropy filter

    ULONG        m_nViewX;           // X Position of render viewport
    ULONG        m_nViewY;           // Y Position of render viewport
    ULONG        m_nViewWidth;       // Width of render viewport
    ULONG        m_nViewHeight;      // Height of render viewport
    POINT        m_OldCursorPos;     // Old cursor position for tracking
    CObject      m_Object;           // The object referencing the player mesh
    CMesh        m_PlayerMesh;       // The player mesh (cube ;)

    CMesh                  m_SkyMesh;        // The skybox mesh (also a cube)
    LPDIRECT3DTEXTURE9     m_SkyTextures[6]; // The skybox textures
};
```

D3DFILLMODE m_FillMode
This variable is controlled via a menu option to switch between **D3DFILL_WIREFRAME** and **D3DFILL_SOLID**.

D3DTEXTUREOP m_ColorOp
This will hold the texture operation used in texture stage 1 when performing single pass multi texturing. By default this is D3DTOP_ADDSIGNED if that color operation is supported on the current hardware. The reason we record the operation is that the user is allowed to switch between color operations.

D3DTEXTUREFILTERTYPE m_MagFilter;
D3DTEXTUREFILTERTYPE m_MinFilter;
D3DTEXTUREFILTERTYPE m_MipFilter;
This application allows the user to select different filter types for the magnification, minification, and MIP filter types. These variables record which filters the user has selected and are currently in use.

ULONG m_Anisotropy;
The application will allow the user to enable anisotropic filtering if supported on the hardware. If anisotropic is enabled, this variable holds the number of levels of anisotropic filtering that the user has currently selected. The higher the level, the more expensive it is, but the better it looks.

bool m_bSinglePass;
If the application is capable of rendering the terrain in a single pass using two texture stages then this Boolean will be set to true. If not, it will be set to false. We test this Boolean during rendering to see if we have to do multiple passes of the terrain or whether we can just assign the two textures to the two stages and render it once.

bool m_MagFilterCaps[10];
bool m_MinFilterCaps[10];
bool m_MipFilterCaps[10];
bool m_ColorOpCaps[30];
These Boolean arrays contain either true or false if the relevant filter or color op is supported. The array element will correspond with the value of the member of the enumerated type. So we would be able to check whether we can do D3DTEXF_LINEAR MIP mapping for example by doing the following:

```
if (m_MipFilterCaps[ D3DTEXF_LINEAR ] ) // We can set linear filtering
```

Likewise, we will be able to check if the application can perform the D3DTOP_MODULATE2X color operation by checking like so:

```
if (m_ColorOpCaps[ D3DTOP_MODULATE2X] ) // We can do modulate2X
```

We will fill out these Boolean arrays in the CGameApp::TestDeviceCaps function.

ULONG m_MaxTextures;
This member will hold the maximum number of simultaneous textures that the hardware can use in a single pass. If this is smaller than 2, then we will have to set m_bSinglePass to false and render our terrain in two separate passes.

ULONG m_MaxAnisotropy;
This will be filled out in the TestDeviceCaps function and will hold the maximum level of anisotropy that the hardware can support. This allows the user to change levels at run time using the menu interface.

CMesh m_SkyMesh;
LPDIRECT3DTEXTURE9 m_SkyTextures[6];
These final two members that we have added to the CGameApp class require some explanation. If you have already run the .exe file from Lab Project 6.2 you will likely have noticed the addition of a sky to our outdoor environment. This looks superior to just rendering our terrain polygons on top of a blue frame buffer as we did in earlier lessons. There are a number of techniques that can be used to create an environment complete with sky, clouds, and even scenery. The approach we used here, called a sky box, is one of the most commonly used and easy to implement.

Sky Boxes

A sky box is a mesh (typically a cube) that completely surrounds the camera at all times. The winding order of the cube faces are such that the face normals of each cube face point inwards towards the camera located at the center of the cube.

The box is situated in our 3D world such that the camera itself is always located directly at its center. Whenever the cameras position us updated, so too is the position of the sky box mesh so that the camera always remains at its center. Note that whilst the sky box mesh has its position in the 3D world synchronized to the position for the camera, the camera is allowed to rotate freely inside the sky box, allowing the camera to look up at the sky or down at the ground. The sky box mesh itself is never rotated and always remains aligned with the world axes.

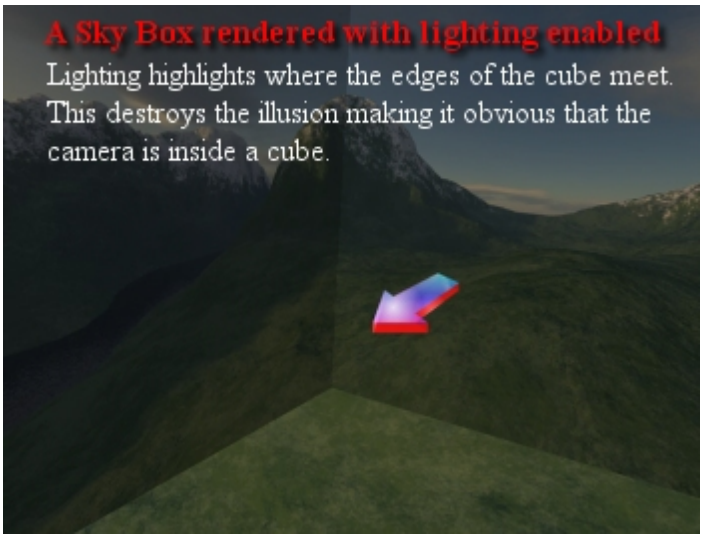
On each of the six faces of the box, a texture is placed. These textures should be seamless so that taken as a whole they depict a full panoramic view of the surrounding environment. This provides an easy way to add distant scenery and realistic atmospheric effects to outdoor environments. The sky box textures can be created offline using any number of rendering packages. TerraGen™ includes an easy to use interface specifically for this purpose. Each texture will be fully mapped and clamped to the appropriate box face when the cube is rendered.

Since the distant scenery depicted in a sky box will never occlude scene objects, sky boxes are generally rendered before any other object in the scene. It is worth noting that the cube geometry that envelops the camera is not itself generally large enough to surround the entire scene in world space. In fact, very often the cube mesh itself is quite small to help minimize texture stretching artifacts. As a result, given that our camera is located ‘inside’ the cube, we do not wish the sky box to occlude our

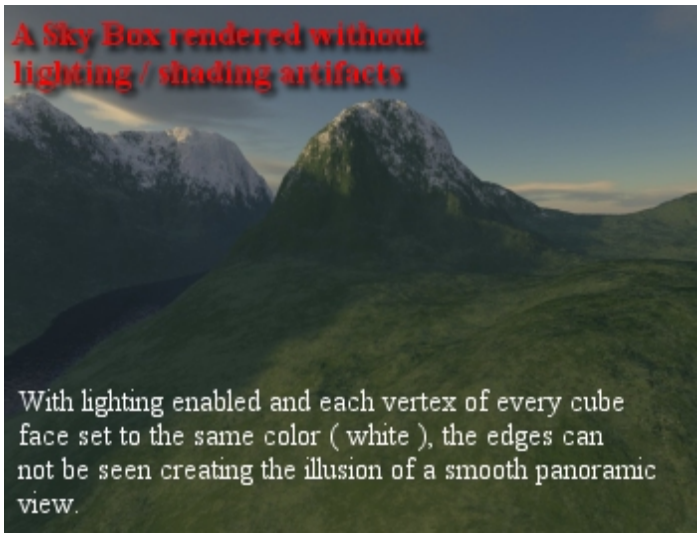
actual 3D scene objects. For example, the cube might be quite small in size such that if rendered in the normal way with depth buffering enabled, the front face of the sky box might be closer to the camera than other objects in our scene that we would always want to appear to be inside it. Therefore depth buffering will be disabled when we render our sky box to avoid pixel depth values entering the buffer that would prevent other object pixels from being seen. Certainly there is little point in performing thousands of per-pixel depth tests when the outcome of those tests is known in advance. This means we will typically render the sky box at the start of our scene rendering function before any other objects are rendered. Because the sky box depth values will not be written to the depth buffer, we can rest assured that regardless of the size of the sky box or the distance between the camera and the sky box faces, all objects rendered thereafter will be rendered 'on top of' the sky box pixels. This creates the illusion that our sky box is a huge scene-encompassing box.

We can think of sky rendering as being more akin to painting the frame buffer. Since the rendering of a sky box completely fills the frame buffer with color, we can actually avoid clearing the frame buffer if we wish. This is a small optimization to be sure, as frame buffer clearing is generally quite rapid, but worth mentioning nonetheless.

One thing we must make sure we do if using the lighting pipeline is disable it before we render the sky box. We will typically use pre-lit vertices for the sky box, such that every vertex in the cube is assigned the same color (usually white). This is because we wish to hide the seams where the cube faces meet. If lighting was enabled prior to rendering the sky box, then the Gouraud shading functionality in the pipeline will calculate the diffuse color of each vertex, taking the position and orientation of the vertex and its normal into account. This may result in slightly different per-vertex colors being calculated. The following image shows what can happen when lighting is enabled. We can see that in the bottom corner of the sky box mesh where the left, right, and bottom faces meet, each has had slightly different per-vertex colors calculated by the lighting pipeline. Although this shading is usually desirable, in the case of a sky box, it emphasizes that the player is inside of a cube, thus destroying the illusion of a smooth panoramic view.



With lighting disabled and identical vertex colors assigned, these edges are no longer visible:



Although we will see later how the sky box is constructed and how it is rendered, we now know enough about what a sky box is to understand the two new members (shown above) that we have added to our CGameApp class. The first is a CMesh object which contains the geometry for the sky box (a cube with faces pointing inwards). The second new member is an array of six IDirect3DTexture9 interfaces. This array will be used to hold the six sky box textures shown in the above images.

CGameApp()

```
CGameApp::CGameApp()
{
   // Reset / Clear all required values
    m_hWnd          = NULL;
    m_pD3D          = NULL;
    m_pD3DDevice    = NULL;
    m_hIcon         = NULL;
    m_hMenu         = NULL;
    m_bLostDevice   = false;
    m_LastFrameRate = 0;

    // Set up initial states (these will be adjusted later if not supported)
    m_FillMode      = D3DFILL_SOLID;
    m_ColorOp       = D3DTOP_ADDSIGNED;
    m_MinFilter     = D3DTEXF_LINEAR;
    m_MagFilter     = D3DTEXF_LINEAR;
    m_MipFilter     = D3DTEXF_LINEAR;
    m_bSinglePass   = true;
    m_Anisotropy    = 1;
}
```

By default, the constructor sets all filters to linear (trilinear filtering) and assigns m_bSinglePass to true because this is our preferred way to render (single pass). We also choose the **D3DTOP_ADDSIGNED** texture stage color operation as our preferred color blending operation in the texture stages. These values may change based on the device capabilities.

CGameApp::InitInstance

This function has not changed much but it might be helpful just to remind ourselves of the program flow.

```
bool CGameApp::InitInstance( HANDLE hInstance, LPCTSTR lpCmdLine, int iCmdShow )
{
    // Create the primary display device
    if (!CreateDisplay()) { ShutDown(); return false; }

    // Build Objects
    if (!BuildObjects()) { ShutDown(); return false; }

    // Set up all required game states
    SetupGameState();

    // Setup our rendering environment
    SetupRenderStates();

    // Success!
    return true;
}
```

CGameApp::SetupRenderStates

The first thing this function does is call the new CGameApp class function TestDeviceCaps to query all of the device states and fill in our filter and color blending arrays.

```
void CGameApp::SetupRenderStates()
{
    // Test the device capabilities.
    if (!TestDeviceCaps( )) { PostQuitMessage(0); return; }
```

Next we disable lighting and set the standard states (enable z-buffer, Gouraud shading, etc.):

```
    // Setup our D3D Device initial states
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, FALSE );
    m_pD3DDevice->SetRenderState( D3DRS_ZENABLE, D3DZB_TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_DITHERENABLE,  TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_SHADEMODE, D3DSHADE_GOURAUD );
    m_pD3DDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CCW );
```

Now we set the default minification, magnification, and MIP filters for stage 0. Although these variables were all set to D3DTEXF_LINEAR by default in the constructor, if linear filtering is not supported, the TestDeviceCaps function called above will have correctly assigned these variables to a default filtering mode that is supported on the current hardware.

```
    // Set up sampler states.
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_MINFILTER    , m_MinFilter );
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_MAGFILTER    , m_MagFilter );
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_MIPFILTER    , m_MipFilter );
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_MAXANISOTROPY, m_Anisotropy);
```

Next we setup the color operation in stage 0. As we have already discussed we will not be using any blending here so we will assign the texture to ARG1 and output the sampled texel directly. We also inform stage 0 that it should use the first set of texture coordinates in the vertex to index into the texture assigned to that stage. Remember that the base texture will be assigned to this stage.

```
    // Set texture / addressing / color ops
    m_pD3DDevice->SetTextureStageState(0,D3DTSS_COLORARG1,D3DTA_TEXTURE);
    m_pD3DDevice->SetTextureStageState(0,D3DTSS_COLOROP,D3DTOP_SELECTARG1);
    m_pD3DDevice->SetTextureStageState(0,D3DTSS_TEXCOORDINDEX,0);
```

The TestDeviceCaps function will have set the m_bSinglePass Boolean to true if the current hardware is capable of using the two texture stages to render the terrain in a single pass. We pass this Boolean to CTerrain::SetRenderMode (a new function) which simply makes a copy of this Boolean so that it knows how to render itself. Remember that CTerrain::Render actually renders the terrain, so it will need to know whether it has to do it in a single pass or multiple passes.

```
    // Inform the terrain of how it should render
    m_Terrain.SetRenderMode( m_bSinglePass );
```

If the device is capable of single pass multi texturing then we can use stage 1 to blend the detail texture during rendering. If this is the case we setup stage 1 so that it blends the output from stage 0 (the base texture) with the texture assigned to stage 1 (the detail texture). We also set the color operation which is stored in the m_ColorOp variable. Preferably it will have been set to **D3DTOP_ADDSIGNED** by the TestDeviceCaps function. If not, then another blending function will be set instead. We also inform stage 1 to index into the detail map using the second set of texture coordinates in the vertex. Finally, we setup the filters for stage 1 as we did with stage 0.

```
    // If we are performing single pass rendering
    if ( m_bSinglePass ){
      m_pD3DDevice->SetTextureStageState(1,D3DTSS_COLORARG1,D3DTA_CURRENT);
      m_pD3DDevice->SetTextureStageState(1,D3DTSS_COLORARG2,D3DTA_TEXTURE);
      m_pD3DDevice->SetTextureStageState(1,D3DTSS_COLOROP, m_ColorOp);
      m_pD3DDevice->SetTextureStageState(1,D3DTSS_TEXCOORDINDEX,1);
      m_pD3DDevice->SetSamplerState(1,D3DSAMP_MINFILTER,m_MinFilter);
      m_pD3DDevice->SetSamplerState(1,D3DSAMP_MAGFILTER,m_MagFilter);
      m_pD3DDevice->SetSamplerState(1,D3DSAMP_MIPFILTER,m_MipFilter);
      m_pD3DDevice->SetSamplerState(1,D3DSAMP_MAXANISOTROPY,m_Anisotropy);
    }
```

```
    else
    {
      // Disable second stage processing
      m_pD3DDevice->SetTextureStageState(1,D3DTSS_COLOROP,D3DTOP_DISABLE);
      m_pD3DDevice->SetTexture( 1, NULL );
    }
```

The else statement is executed if the m_bSinglePass Boolean is false and we have to use multiple passes. Therefore, we make sure that the texture stage is disabled and any texture that may be assigned to it cleared before proceeding.

Next, we set the fill mode and FVF flags and update the camera settings:

```
    // Setup option dependant states
    m_pD3DDevice->SetRenderState( D3DRS_FILLMODE, m_FillMode );

    // Setup our default vertex FVF code
    m_pD3DDevice->SetFVF( VERTEX_FVF );

    // Update our device with our camera details (Required on reset)
    if ( !m_pCamera ) return;
    m_pCamera->UpdateRenderView( m_pD3DDevice );
    m_pCamera->UpdateRenderProj( m_pD3DDevice );
```

Finally we call SelectMenuItems to enable or disable menu items that are available or not.

```
    // Set up the menu item selections
    //(Which may have changed during device validations)
    SelectMenuItems();
}
```

Recall that this function will be called when the device is changed by the user. So if the user decides to change from a windowed device to a fullscreen device, the capabilities may well have changed for the fullscreen device.

CGameApp::TestDeviceCaps

TestDeviceCaps checks the device capabilities and fills out the available filter and color op arrays. Since it includes a lot of repetitive code we will look only at the first section.

The first thing we do is get the device caps and clear the CGameApp filter and color operation arrays.

```
bool CGameApp::TestDeviceCaps( )
{
    HRESULT  hRet;
    D3DCAPS9 Caps;
    ULONG    Enable, Value;

    // Retrieve device caps
    hRet = m_pD3D->GetDeviceCaps(
```

```
                    m_D3DSettings.GetSettings()->AdapterOrdinal,
                    m_D3DSettings.GetSettings()->DeviceType, &Caps );
    if ( FAILED(hRet) ) return false;

    // Reset our caps storage.
    // Note: These store the available of actual state values
    ZeroMemory( m_MinFilterCaps   , 10 * sizeof(bool) );
    ZeroMemory( m_MagFilterCaps   , 10 * sizeof(bool) );
    ZeroMemory( m_MipFilterCaps   , 10 * sizeof(bool) );
    ZeroMemory( m_ColorOpCaps     , 30 * sizeof(bool) );

    // Set up those states always supported
    m_MinFilterCaps[D3DTEXF_NONE] = true;
    m_MagFilterCaps[D3DTEXF_NONE] = true;
    m_MipFilterCaps[D3DTEXF_NONE] = true;
    m_MaxTextures   = 0;
    m_MaxAnisotropy = 0;
```

Now we will test the texture filtering capabilities of the device. These are stored in the D3DCAPS9::TextureFilterCaps member. We can test the bits to see if anisotropic, linear, or point minification filters are supported. For each one that has its bit set, we set its corresponding Boolean to true in the m_minFilterCaps array. We also enable that corresponding menu item in the application menu.

```
    // Test Texture Filter Caps
    Value = Caps.TextureFilterCaps;

    // Determine if anisotropic minification filtering is supported
    Enable = MF_ENABLED;
    if(!(Value & D3DPTFILTERCAPS_MINFANISOTROPIC)) Enable = MF_DISABLED | MF_GRAYED;
    EnableMenuItem( m_hMenu, ID_MINFILTER_ANISOTROPIC, MF_BYCOMMAND | Enable );
    if( Enable == MF_ENABLED ) m_MinFilterCaps[D3DTEXF_ANISOTROPIC] = true;

    // Determine if linear minification filtering is supported
    Enable = MF_ENABLED;
    if(!(Value & D3DPTFILTERCAPS_MINFLINEAR)) Enable = MF_DISABLED | MF_GRAYED;
    EnableMenuItem( m_hMenu, ID_MINFILTER_LINEAR, MF_BYCOMMAND | Enable );
    if( Enable == MF_ENABLED ) m_MinFilterCaps[D3DTEXF_LINEAR] = true;

    // Determine if point minification filtering is supported
    Enable = MF_ENABLED;
    if(!(Value & D3DPTFILTERCAPS_MINFPOINT)) Enable = MF_DISABLED | MF_GRAYED;
    EnableMenuItem( m_hMenu, ID_MINFILTER_POINT, MF_BYCOMMAND | Enable );
    if( Enable == MF_ENABLED ) m_MinFilterCaps[D3DTEXF_POINT] = true;
```

Now we repeat the process for magnification filters. If we find support for a filter mode we set the corresponding Boolean in the CGameApp::m_MagFilterCaps array to true and enable the relevant menu item.

```
    // Determine if anisotropic magnification filtering is supported
    Enable = MF_ENABLED;
    if(!(Value & D3DPTFILTERCAPS_MAGFANISOTROPIC))
          Enable = MF_DISABLED | MF_GRAYED;
    EnableMenuItem( m_hMenu, ID_MAGFILTER_ANISOTROPIC, MF_BYCOMMAND | Enable );
```

```
    if( Enable == MF_ENABLED ) m_MagFilterCaps[D3DTEXF_ANISOTROPIC] = true;

    // Determine if linear magnification filtering is supported
    Enable = MF_ENABLED;
    if(!(Value & D3DPTFILTERCAPS_MAGFLINEAR)) Enable = MF_DISABLED | MF_GRAYED;
    EnableMenuItem( m_hMenu, ID_MAGFILTER_LINEAR, MF_BYCOMMAND | Enable );
    if( Enable == MF_ENABLED ) m_MagFilterCaps[D3DTEXF_LINEAR] = true;

    // Determine if point magnification filtering is supported
    Enable = MF_ENABLED;
    if(!(Value & D3DPTFILTERCAPS_MAGFPOINT)) Enable = MF_DISABLED | MF_GRAYED;
    EnableMenuItem( m_hMenu, ID_MAGFILTER_POINT, MF_BYCOMMAND | Enable );
    if( Enable == MF_ENABLED ) m_MagFilterCaps[D3DTEXF_POINT] = true;
```

Finally, we test the MIP filtering methods and record the results in the m_MipFilterCaps array. Anisotropic is not a valid MIP filter so we check for only Point and Linear filtering methods.

```
    // Determine if linear mip filtering is supported
    Enable = MF_ENABLED;
    if ( !(Value & D3DPTFILTERCAPS_MIPFLINEAR) )
         Enable = MF_DISABLED | MF_GRAYED;
    EnableMenuItem( m_hMenu, ID_MIPFILTER_LINEAR, MF_BYCOMMAND | Enable );
    if ( Enable == MF_ENABLED ) m_MipFilterCaps[D3DTEXF_LINEAR] = true;

    // Determine if point mip filtering is supported
    Enable = MF_ENABLED;
    if ( !(Value & D3DPTFILTERCAPS_MIPFPOINT) )
         Enable = MF_DISABLED | MF_GRAYED;
    EnableMenuItem( m_hMenu, ID_MIPFILTER_POINT, MF_BYCOMMAND | Enable );
    if ( Enable == MF_ENABLED ) m_MipFilterCaps[D3DTEXF_POINT] = true;
```

Next we check supported color operations. While we will not be requiring all of them, we will provide a nice selection of color operations to choose from. This enables us to see the effect of the different blending parameters between the detail and base textures. To check the various texture state blending operations we need to check the bits of the D3DCAPS9::TextureOpCaps member. If one of the color operations we are testing is supported, we will enable its menu item and set the relevant Boolean to true in the CGameApp::m_ColorOpCaps array.

```
    // Test texture operation caps
    Value = Caps.TextureOpCaps;

    // Determine if 'Add Signed' op is supported
    Enable = MF_ENABLED;
    if ( !(Value & D3DTEXOPCAPS_ADDSIGNED) ) Enable = MF_DISABLED | MF_GRAYED;
    EnableMenuItem( m_hMenu, ID_COLOROP_ADDSIGNED, MF_BYCOMMAND | Enable );
    if ( Enable == MF_ENABLED ) m_ColorOpCaps[D3DTOP_ADDSIGNED] = true;

    // Determine if 'Modulate 2x' op is supported
    Enable = MF_ENABLED;
    if ( !(Value & D3DTEXOPCAPS_MODULATE2X) ) Enable = MF_DISABLED | MF_GRAYED;
    EnableMenuItem( m_hMenu, ID_COLOROP_MODULATE2X, MF_BYCOMMAND | Enable );
    if ( Enable == MF_ENABLED ) m_ColorOpCaps[D3DTOP_MODULATE2X] = true;

    // Determine if 'Modulate' op is supported
    Enable = MF_ENABLED;
```

```
    if ( !(Value & D3DTEXOPCAPS_MODULATE) ) Enable = MF_DISABLED | MF_GRAYED;
    EnableMenuItem( m_hMenu, ID_COLOROP_MODULATE, MF_BYCOMMAND | Enable );
    if ( Enable == MF_ENABLED ) m_ColorOpCaps[D3DTOP_MODULATE] = true;

    // Determine if 'Add' op is supported
    Enable = MF_ENABLED;
    if ( !(Value & D3DTEXOPCAPS_ADD) ) Enable = MF_DISABLED | MF_GRAYED;
    EnableMenuItem( m_hMenu, ID_COLOROP_ADD, MF_BYCOMMAND | Enable );
    if ( Enable == MF_ENABLED ) m_ColorOpCaps[D3DTOP_ADD] = true;

    // Determine if 'Add Signed 2x' op is supported
    Enable = MF_ENABLED;
    if ( !(Value & D3DTEXOPCAPS_ADDSIGNED2X) )
    Enable = MF_DISABLED | MF_GRAYED;
    EnableMenuItem( m_hMenu, ID_COLOROP_ADDSIGNED2X, MF_BYCOMMAND | Enable );
    if ( Enable == MF_ENABLED ) m_ColorOpCaps[D3DTOP_ADDSIGNED2X] = true;

    // Determine if 'Modulate 4x' op is supported
    Enable = MF_ENABLED;
    if ( !(Value & D3DTEXOPCAPS_MODULATE4X) ) Enable = MF_DISABLED | MF_GRAYED;
    EnableMenuItem( m_hMenu, ID_COLOROP_MODULATE4X, MF_BYCOMMAND | Enable );
    if ( Enable == MF_ENABLED ) m_ColorOpCaps[D3DTOP_MODULATE4X] = true;

    // Determine if 'Subtract' op is supported
    Enable = MF_ENABLED;
    if ( !(Value & D3DTEXOPCAPS_SUBTRACT) ) Enable = MF_DISABLED | MF_GRAYED;
    EnableMenuItem( m_hMenu, ID_COLOROP_SUBTRACT, MF_BYCOMMAND | Enable );
    if ( Enable == MF_ENABLED ) m_ColorOpCaps[D3DTOP_SUBTRACT] = true;
```

Our next task is to query the device to see if the device can handle two simultaneous textures. If it can, then we will enable the menu choice to select between multi-pass and single-pass rendering. If the device does not support at least two textures, then we must disable the menu choice for single-pass. We get this information by querying the D3DCAPS9::MaxSimultaneousTextures variable.

```
    // Determine if single-pass 2 stage texturing is supported
    Enable = MF_ENABLED;
    if ( Caps.MaxSimultaneousTextures < 2 ) Enable = MF_DISABLED | MF_GRAYED;
    EnableMenuItem( m_hMenu, ID_RENDERMODE_SINGLEPASS, MF_BYCOMMAND | Enable );
    if ( Enable == MF_ENABLED ) m_MaxTextures = Caps.MaxSimultaneousTextures;
```

The next section of code queries the D3DCAPS9::MaxAnisotropy member to retrieve the maximum level of anisotropy on the device. Our application menu allows anisotropy levels that start at 1 and increase in powers of 2. The following code enables the menu items that are less than or equal to max anisotropy:

```
    // Test anisotropy levels
    Value = Caps.MaxAnisotropy;

    // Determine which anisotropy levels are supported
    if ( Value < 1 )
       Enable = MF_DISABLED | MF_GRAYED; else Enable = MF_ENABLED;
    EnableMenuItem( m_hMenu, ID_MAXANISOTROPY_1, MF_BYCOMMAND | Enable );
    if ( Enable == MF_ENABLED ) m_MaxAnisotropy = 1;
```

```
    if ( Value < 2 )
       Enable = MF_DISABLED | MF_GRAYED; else Enable = MF_ENABLED;
    EnableMenuItem( m_hMenu, ID_MAXANISOTROPY_2, MF_BYCOMMAND | Enable );
    if ( Enable == MF_ENABLED ) m_MaxAnisotropy = 2;

    if ( Value < 4 )
       Enable = MF_DISABLED | MF_GRAYED; else Enable = MF_ENABLED;
    EnableMenuItem( m_hMenu, ID_MAXANISOTROPY_4, MF_BYCOMMAND | Enable );
    if ( Enable == MF_ENABLED ) m_MaxAnisotropy = 4;

    if ( Value < 8 )
        Enable = MF_DISABLED | MF_GRAYED; else Enable = MF_ENABLED;
    EnableMenuItem( m_hMenu, ID_MAXANISOTROPY_8, MF_BYCOMMAND | Enable );
    if ( Enable == MF_ENABLED ) m_MaxAnisotropy = 8;

    if ( Value < 16 ) Enable = MF_DISABLED | MF_GRAYED;
    else Enable = MF_ENABLED;
    EnableMenuItem( m_hMenu, ID_MAXANISOTROPY_16, MF_BYCOMMAND | Enable );
    if ( Enable == MF_ENABLED ) m_MaxAnisotropy = 16;

    if ( Value < 32 ) Enable = MF_DISABLED | MF_GRAYED;
    else Enable = MF_ENABLED;
    EnableMenuItem( m_hMenu, ID_MAXANISOTROPY_32, MF_BYCOMMAND | Enable );
    if ( Enable == MF_ENABLED ) m_MaxAnisotropy = 32;

    if ( Value < 64 ) Enable = MF_DISABLED | MF_GRAYED;
    else Enable = MF_ENABLED;
    EnableMenuItem( m_hMenu, ID_MAXANISOTROPY_64, MF_BYCOMMAND | Enable );
    if ( Enable == MF_ENABLED ) m_MaxAnisotropy = 64;
```

Finally, we need to set our default modes. Remember, although the m_MinFactor variable may hold the default desired minification filter set in the class constructor, it may not be supported on the hardware. Furthermore, if the user has switched devices, the filters may not be applicable on the new device. This function is called every time the device is built to reset menu items and assign default settings that work with the device. We test to see if the currently set minification filter is supported by checking the Boolean in the minification array that we have just filled out earlier in the function. If it is not supported, then we start at the top and search for one that has its corresponding Boolean set to true in the array.

```
    // Now determine if our currently selected states are supported, swap otherwise
    if ( m_MinFilterCaps[ m_MinFilter ] == false )
    {
        if ( m_MinFilterCaps[ D3DTEXF_ANISOTROPIC ] )
           m_MinFilter = D3DTEXF_ANISOTROPIC;
        else if ( m_MinFilterCaps[ D3DTEXF_LINEAR ] )
           m_MinFilter = D3DTEXF_LINEAR;
        else if ( m_MinFilterCaps[ D3DTEXF_POINT ] ) m_MinFilter = D3DTEXF_POINT;
        else if ( m_MinFilterCaps[ D3DTEXF_NONE  ] ) m_MinFilter = D3DTEXF_NONE;
        else return false;
    }
```

We do the same thing for the magnification filter and the MIP filter.

```
    if ( m_MagFilterCaps[ m_MagFilter ] == false )
    {
        if ( m_MagFilterCaps[ D3DTEXF_ANISOTROPIC ] )
           m_MagFilter = D3DTEXF_ANISOTROPIC;
        else if ( m_MagFilterCaps[ D3DTEXF_LINEAR ] )
           m_MagFilter = D3DTEXF_LINEAR;
        else if ( m_MagFilterCaps[ D3DTEXF_POINT ] ) m_MagFilter = D3DTEXF_POINT;
        else if ( m_MagFilterCaps[ D3DTEXF_NONE  ] ) m_MagFilter = D3DTEXF_NONE;
        else return false;

    } // End if Filter not supported

    if ( m_MipFilterCaps[ m_MipFilter ] == false )
    {
        if ( m_MipFilterCaps[ D3DTEXF_ANISOTROPIC ] ) m_MipFilter = D3DTEXF_ANISOTROPIC;
        else if ( m_MipFilterCaps[ D3DTEXF_LINEAR ] ) m_MipFilter = D3DTEXF_LINEAR;
        else if ( m_MipFilterCaps[ D3DTEXF_POINT ] )  m_MipFilter = D3DTEXF_POINT;
        else if ( m_MipFilterCaps[ D3DTEXF_NONE  ] )  m_MipFilter = D3DTEXF_NONE;
        else return false;

    } // End if Mip-Filter not supported
```

Finally we employ the same technique to find a default color operation. If the current one is not supported (D3DTOP_ADDSIGNED from the constructor) we step through the color operations in preferred order to find one that is:

```
    if ( m_ColorOpCaps[ m_ColorOp ] == false )
    {
        if ( m_ColorOpCaps[ D3DTOP_ADDSIGNED ] )      m_ColorOp = D3DTOP_ADDSIGNED;
        else if ( m_ColorOpCaps[ D3DTOP_MODULATE2X  ] ) m_ColorOp = D3DTOP_MODULATE2X;
        else if ( m_ColorOpCaps[ D3DTOP_MODULATE    ] ) m_ColorOp = D3DTOP_MODULATE;
        else if ( m_ColorOpCaps[ D3DTOP_ADD         ] ) m_ColorOp = D3DTOP_ADD;
        else if ( m_ColorOpCaps[ D3DTOP_ADDSIGNED2X ] ) m_ColorOp = D3DTOP_ADDSIGNED2X;
        else if ( m_ColorOpCaps[ D3DTOP_MODULATE4X  ] ) m_ColorOp = D3DTOP_MODULATE4X;
        else if ( m_ColorOpCaps[ D3DTOP_SUBTRACT    ] ) m_ColorOp = D3DTOP_SUBTRACT;
        else return false;
    } // End if ColorOp not supported
```

If we determined that the maximum number of simultaneous textures was less than two, we set the m_bSinglePass Boolean to false. This informs the application and the terrain class that it must render the terrain in multiple passes.

```
    // Test for single pass capabilities.
    if ( m_MaxTextures < 2 ) m_bSinglePass = false;

    // Test max anisotropy
    if ( m_Anisotropy > m_MaxAnisotropy ) m_Anisotropy = m_MaxAnisotropy;

    // Success!!
    return true;
}
```

CGameApp::BuildObjects

The 'BuildObjects' function is virtually unchanged from the previous terrain demos we have written. It basically calls the 'CTerrain' objects 'LoadHeightMap' function to construct the terrain and then builds the little 'Cube' that we use as the player mesh. There is however an additional line at the end of the function that calls a brand new 'CGameApp' member function. This function (called 'BuildSkyBox') builds the 6 cube faces of the Sky Box mesh and loads the six textures for the sky box.

```
    // Build the skybox
    if ( !BuildSkyBox() ) return false;
```

CGameApp::BuildSkyBox

The following code should be self explanatory to you by now with your knowledge of creating vertex data and adding that data to a CMesh object. In this example, we make room in the 'Sky Box' mesh for 24 vertices (6 faces * 4 vertices). We then add the vertices for each quad one at a time to the meshes temporary vertex array. You will see if you plot our vertex positions out on paper that this defines a cube that has faces that have a clockwise winding order when inside the cube. This is quite an important point because usually the winding order of a cube face would be defined such that it would be back face culled with the camera placed inside it. We reverse the winding here so that is not the case.

Once the vertices have been added we then generate the indices such that the cube is described as an array of triangles (as we know must be the case). We then call the CMesh::BuildBuffers member function so that the actual index and vertex buffers of the CMesh get generated using our newly generated index and vertex data. We end the function by loading the 6 Sky Box textures into the 'CGameApp::m_SkyTextures' array.

```
bool CGameApp::BuildSkyBox()
{
    HRESULT hRet;

    // Allocate our 24 mesh vertices
    m_SkyMesh.SetVertexFormat( VERTEX_FVF, sizeof(CVertex) );
    if ( m_SkyMesh.AddVertex( 24 ) < 0 ) return false;

    // Build the skybox vertices
    CVertex * pVertices = (CVertex*)m_SkyMesh.m_pVertex;

    // Front quad (remember all quads point inward)
    pVertices[0]  = CVertex( -10.0f,  10.0f,  10.0f, 0.0f, 0.0f );
    pVertices[1]  = CVertex(  10.0f,  10.0f,  10.0f, 1.0f, 0.0f );
    pVertices[2]  = CVertex(  10.0f, -10.0f,  10.0f, 1.0f, 1.0f );
    pVertices[3]  = CVertex( -10.0f, -10.0f,  10.0f, 0.0f, 1.0f );

    // Back Quad
    pVertices[4]  = CVertex(  10.0f,  10.0f, -10.0f, 0.0f, 0.0f );
```

```
    pVertices[5]  = CVertex( -10.0f,  10.0f, -10.0f, 1.0f, 0.0f );
    pVertices[6]  = CVertex( -10.0f, -10.0f, -10.0f, 1.0f, 1.0f );
    pVertices[7]  = CVertex(  10.0f, -10.0f, -10.0f, 0.0f, 1.0f );

    // Left Quad
    pVertices[8]  = CVertex( -10.0f,  10.0f, -10.0f, 0.0f, 0.0f );
    pVertices[9]  = CVertex( -10.0f,  10.0f,  10.0f, 1.0f, 0.0f );
    pVertices[10] = CVertex( -10.0f, -10.0f,  10.0f, 1.0f, 1.0f );
    pVertices[11] = CVertex( -10.0f, -10.0f, -10.0f, 0.0f, 1.0f );

    // Right Quad
    pVertices[12] = CVertex(  10.0f,  10.0f,  10.0f, 0.0f, 0.0f );
    pVertices[13] = CVertex(  10.0f,  10.0f, -10.0f, 1.0f, 0.0f );
    pVertices[14] = CVertex(  10.0f, -10.0f, -10.0f, 1.0f, 1.0f );
    pVertices[15] = CVertex(  10.0f, -10.0f,  10.0f, 0.0f, 1.0f );

    // Top Quad
    pVertices[16] = CVertex( -10.0f,  10.0f, -10.0f, 0.0f, 0.0f );
    pVertices[17] = CVertex(  10.0f,  10.0f, -10.0f, 1.0f, 0.0f );
    pVertices[18] = CVertex(  10.0f,  10.0f,  10.0f, 1.0f, 1.0f );
    pVertices[19] = CVertex( -10.0f,  10.0f,  10.0f, 0.0f, 1.0f );

    // Bottom Quad
    pVertices[20] = CVertex( -10.0f, -10.0f,  10.0f, 0.0f, 0.0f );
    pVertices[21] = CVertex(  10.0f, -10.0f,  10.0f, 1.0f, 0.0f );
    pVertices[22] = CVertex(  10.0f, -10.0f, -10.0f, 1.0f, 1.0f );
    pVertices[23] = CVertex( -10.0f, -10.0f, -10.0f, 0.0f, 1.0f );

    // Allocate the indices
    if ( m_SkyMesh.AddIndex( 24 ) < 0 ) return false;
    USHORT * pIndices = m_SkyMesh.m_pIndex;

    // Set the indices for each face (tri strip)
    for ( ULONG i = 0; i < 6; i++ )
    {
        // Build the skybox indices
        *pIndices++ = (i*4);
        *pIndices++ = (i*4) + 1;
        *pIndices++ = (i*4) + 3;
        *pIndices++ = (i*4) + 2;

    } // Next 'face'

    VERTEXPROCESSING_TYPE vp=m_D3DSettings.GetSettings()->VertexProcessingType;
    bool HardwareTnL = true;

    // Are we using HardwareTnL ?
    if ( vp != HARDWARE_VP && vp != PURE_HARDWARE_VP ) HardwareTnL = false;

    // Build the mesh buffers
    if ( FAILED(m_SkyMesh.BuildBuffers( m_pD3DDevice, HardwareTnL ))) return false;

    // Load Textures
    hRet  = D3DXCreateTextureFromFile( m_pD3DDevice,
                                       _T("Data\\SkyBox_Front.jpg") , &m_SkyTextures[0] );
    hRet |= D3DXCreateTextureFromFile( m_pD3DDevice,
                                       _T("Data\\SkyBox_Back.jpg")  , &m_SkyTextures[1] );
    hRet |= D3DXCreateTextureFromFile( m_pD3DDevice,
                                       _T("Data\\SkyBox_Left.jpg")  , &m_SkyTextures[2] );
```

```
    hRet |= D3DXCreateTextureFromFile( m_pD3DDevice,
                                       _T("Data\\SkyBox_Right.jpg") , &m_SkyTextures[3] );
    hRet |= D3DXCreateTextureFromFile( m_pD3DDevice,
                                       _T("Data\\SkyBox_Top.jpg")   , &m_SkyTextures[4] );
    hRet |= D3DXCreateTextureFromFile( m_pD3DDevice,
                                       _T("Data\\SkyBox_Bottom.jpg"), &m_SkyTextures[5] );
    if ( FAILED(hRet) ) return false;

    // Success!!
    return true;
}
```

CGameApp::FrameAdvance

The FrameAdvance function has also not changed since the last demo but we include it simply to remind you of the program flow. The only addition to this function which is worthy of note is that before we render the terrain we now call the CGameApp::RenderSkyBox function. This function (which we will cover momentarily) will synchronize the position of the Sky Box mesh with the camera and render it with depth buffering disabled.

```
    // Clear the frame & depth buffer ready for drawing
    m_pD3DDevice->Clear(0,NULL,D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,0x79D3FF,1.0f,0);

    // Begin Scene Rendering
    m_pD3DDevice->BeginScene();

    // Reset our world matrix
    m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_mtxIdentity );

    // Render the sky box
    RenderSkyBox( );

    // Render our terrain objects
    m_Terrain.Render( );

    // Request our player render itselfs
    m_Player.Render( m_pD3DDevice );

    // End Scene Rendering
    m_pD3DDevice->EndScene();

    // Present the buffer
    if(FAILED(m_pD3DDevice->Present(NULL,NULL,NULL,NULL)))m_bLostDevice = true;
```

The final member function of the CGameApp class that we need to cover is the ‘RenderSkyBox’ function . This is called before we render any objects in our so that it paints the frame buffer with the image of our sky as viewed from the camera.

CGameApp::RenderSkyBox

The first thing this function does (after validating that the device and the camera are valid objects) is get the current world space position of the camera. This is because we need to update the position of the Sky Box mesh too so that it remains synchronized with the camera. We then use the cameras position to build a world matrix that will be used to render the faces of the sky box. Notice how we slightly offset the position of the Sky Box center point 1.3 units from the cameras position along the Y axis. The only reason we did this is because is it looked slightly better that way. Once the world matrix has be built for the Sky Box we make sure that we disable lighting (just in case it was enabled for some reason) and disable Z Writing so that the sky boxes depth values are not written to the depth buffer when rendered. This is important because we do not want our Sky Box to occlude any object in our scene rendered thereafter. Finally, we set the texture addressing mode for stage 0 (the stage the sky box textures will be bound too) to D3DTADDRESS_CLAMP. This is another important step because we need to make sure that no seams appear between our cube faces. Because we are using filtering, the texels along the edge of each texture will be averaged with surrounding texels. This will also cause seams so we want to make sure that the pixels at the edge of each texture are rendered to the screen exactly 'as is' so that the textures on each face line up correctly.

```
void CGameApp::RenderSkyBox( )
{
    D3DXMATRIX mtxWorld;
    D3DXMatrixIdentity( &mtxWorld );

    // Validate parameters
    if ( !m_pCamera || !m_pD3DDevice ) return;

    // Generate our sky box rendering origin and set as world matrix
    D3DXVECTOR3 CamPos = m_pCamera->GetPosition();
    D3DXMatrixTranslation( &mtxWorld, CamPos.x, CamPos.y + 1.3f, CamPos.z );
    m_pD3DDevice->SetTransform( D3DTS_WORLD, &mtxWorld );

    // Set up rendering states for the sky box
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, FALSE );
    m_pD3DDevice->SetRenderState( D3DRS_ZWRITEENABLE, FALSE );
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_ADDRESSU, D3DTADDRESS_CLAMP );
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_ADDRESSV, D3DTADDRESS_CLAMP );
```

If we are using single pass multi-texturing then we must make sure that we disable the second stage as we will only be using stage zero to render the Sky Box. Following the disabling of texture stage one we then set the FVF of our sky box mesh and bind the index buffer and the vertex buffer of the mesh to the device so that it will be used in the following 'DrawPrimitive' calls. When then loop through each of the 6 faces of our cube, setting the appropriate texture and rendering the appropriate face. As each face is basically a quad, we now we can access each one using the 'I*4' calculation.

```
  // Disable second stage if enabled
  if ( m_bSinglePass )
      m_pD3DDevice->SetTextureStageState( 1, D3DTSS_COLOROP, D3DTOP_DISABLE );

  // Render the sky box
```

```
    m_pD3DDevice->SetFVF( m_SkyMesh.m_nFVFCode );
    m_pD3DDevice->SetIndices( m_SkyMesh.m_pIndexBuffer );
    m_pD3DDevice->SetStreamSource( 0, m_SkyMesh.m_pVertexBuffer, 0, m_SkyMesh.m_nStride );

    // Render the 6 sides of the skybox
    for ( ULONG i = 0; i < 6; ++i )
    {
        m_pD3DDevice->SetTexture( 0, m_SkyTextures[i] );
        m_pD3DDevice->DrawIndexedPrimitive( D3DPT_TRIANGLESTRIP, 0, 0, 24, i * 4, 2 );

    } // Next side

    // Reset our states
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_ADDRESSU, D3DTADDRESS_WRAP );
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_ADDRESSV, D3DTADDRESS_WRAP );
    m_pD3DDevice->SetRenderState( D3DRS_ZWRITEENABLE, TRUE );
    m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_mtxIdentity );

    // Re-enable second stage if enabled
    if ( m_bSinglePass )
        m_pD3DDevice->SetTextureStageState( 1, D3DTSS_COLOROP, m_ColorOp );
}
```

Finally, we reset the default states that were changed by this function. We set the texture addressing modes back to D3DTADDRESS_WRAP, enable Z-Writing and re-enable the second texture stage is the application is going to use multiple texture stages to render the terrain in a single pass.

The CTerrain Class

The CTerrain class has three new members. It now contains a Boolean variable specifying whether the terrain should use two texture stages or must be rendered using multiple passes. This variable was set from the CGameApp::SetupRenderStates function with a call to the CTerrain::SetRenderMode in response to querying whether the device could handle two textures simultaneously. We also need pointers to two IDirect3DTexture9 interfaces for the two textures used by the terrain.

```
bool                m_bSinglePass;      // Use single pass rendering method?
LPDIRECT3DTEXTURE9  m_pBaseTexture;     // Base terrain texture
LPDIRECT3DTEXTURE9  m_pDetailTexture;   // Terrain detail texture
```

CTerrain::LoadHeightMap

The CTerrain::LoadHeightMap now includes two calls to D3DXCreateTextureFromFile to load texture images for the base map and the detail map. Below we see the LoadHeightMap code with the two new lines.

```
bool CTerrain::LoadHeightMap( LPCTSTR FileName, ULONG Width, ULONG Height )
{
    HRESULT hRet;
    FILE  * pFile = NULL;
```

```
    // First of all store the information passed
    m_nHeightMapWidth  = Width;
    m_nHeightMapHeight = Height;

    // A scale of 4 is roughly the best size for a 512 x 512 quad terrain.
    // Using the following forumla, lowering the size of the terrain
    // simply lowers the vertex resolution but maintains the map size.
    m_vecScale.x = 4.0f * (512 / (m_nHeightMapWidth - 1));
    m_vecScale.y = 2.0f;
    m_vecScale.z = 4.0f * (512 / (m_nHeightMapHeight - 1));

    // Attempt to allocate space for this heightmap information
    m_pHeightMap = new UCHAR[Width * Height];

    // Open up the heightmap file
    pFile = _tfopen( FileName, _T("rb") );

    // Read the heightmap data (Read only 'Red' component)
    for ( ULONG i = 0; i < Width * Height; i++ )
    {
        fread( &m_pHeightMap[i], 1, 1, pFile );
        fseek( pFile, 2, SEEK_CUR );
    }

    // Finish up
    fclose( pFile );

    // Load in the textures used for rendering the terrain
    D3DXCreateTextureFromFile( m_pD3DDevice, BaseTextureName, &m_pBaseTexture );
    D3DXCreateTextureFromFile( m_pD3DDevice, DetailTextureName, &m_pDetailTexture );

    // Allocate enough meshes to store the separate blocks of this terrain
    if ( AddMesh( ((Width - 1) / QuadsWide) * ((Height - 1) / QuadsHigh) ) < 0 )
        return false;
    // Build the mesh data itself
    return BuildMeshes( );
}
```

The D3DXCreateTextureFromFile function loads the images into the D3DPOOL_MANAGED memory pool and generates a complete MIP chain of filtered surfaces. Remember that the AddMesh function takes care of adding a number of CMesh objects to the CTerrain mesh array (see Chapter 3).

CTerrain::BuildMeshes

We will examine only a very small section of this function since it is virtually unchanged. The main difference now is the calculation of texture coordinates for the base and detail textures.

```
for ( vz = StartZ; vz < StartZ + BlockHeight; vz++ )
{
    for ( vx = StartX; vx < StartX + BlockWidth; vx++ )
    {
        // Calculate and Set The vertex data.
        pVertex[Counter].x  = (float)vx * m_vecScale.x;
```

```
        pVertex[Counter].y  = (float)m_pHeightMap[vx + vz * m_nHeightMapWidth]\
                              * m_vecScale.y;
        pVertex[Counter].z  = (float)vz * m_vecScale.z;

        pVertex[Counter].tu = (float)vx / (m_nHeightMapWidth - 1);
        pVertex[Counter].tv = (float)vz / (m_nHeightMapHeight - 1);
        pVertex[Counter].tu2 = (float)vx / 6.0f;
        pVertex[Counter].tv2 = (float)vz / 6.0f;
        Counter++;

   } // Next Vertex Column

} // Next Vertex Row
```

Recall that this process is repeated for each 17x17 terrain block. For each block in a row, StartX is increased by 17. For each row, StartZ is increased by 17. This means that for any block, *vz* and *vx* define the offset of that vertex in the height map and *vx*m_vecScale.x* and *vz*m_vecScale.z* describe the actual world space position of the vertex along the X and Z axes. This means that *vx* will always describe the number of the vertex in the row and *vz* will describe the number of the row the vertex is in (in the entire terrain). If we divide *vx* by the total number of vertices the terrain has in a row, we get the U coordinate between 0.0 and 1.0. This describes the exact U coordinate for the base map. If we do the same for *vz* and divide this by the total number of terrain rows, we get a V coordinate in the range of 0.0 to 1.0.

Calculating the second set of texture coordinates is even easier. We simply divide the *vx* and *vz* vertex positions by 6. This means that we are mapping the entire detail texture to each 6x6 quad square. This tiles the detail texture across the terrain just less than 43 times in each dimension. The choice to device by 6 was determined by trial and error so feel free to experiment with this value until you have something that suits your own tastes.

Our vertices now have the texture coordinates they need for both stage 0 and stage 1. The base and detail texture maps will now be mapped onto our quads correctly.

CTerrain::Render

```
void CTerrain::Render( )
{
    ULONG i;

    // Validate parameters
    if( !m_pD3DDevice ) return;
```

First we assign the base texture to texture stage 0 and assign the detail texture to stage 1. Notice how we only assign the detail texture to stage 1 if the device supports multi-texturing. If not, we cannot use stage 1 and will have to switch textures in stage 0 later to render the terrain with a second pass.

```
    // Set our base texture in stage 0
    m_pD3DDevice->SetTexture( 0, m_pBaseTexture );

    // Set detail texture in stage 1 if supported
    if ( m_bSinglePass ) m_pD3DDevice->SetTexture( 1, m_pDetailTexture );
```

We set the FVF flags used by the terrain vertices so that the device object knows to expect two sets of texture coordinates.

```
    // Set the FVF code for the terrain meshes, these will always
    // be identical between each mesh stored here, so we can simply
    // use the first.
    if ( m_nMeshCount > 0 ) m_pD3DDevice->SetFVF( m_pMesh[0]->m_nFVFCode );
```

Remember that at this point the texture stage operations have already been set up by the CGameApp::SetupRenderState function. Therefore, we can simply render the terrain as normal.

```
    // Render Each terrain Mesh
    for ( i = 0; i < m_nMeshCount; i++ )
    {
        // Set the stream sources
        m_pD3DDevice->SetStreamSource( 0, m_pMesh[i]->m_pVertexBuffer, 0,
                                       m_pMesh[i]->m_nStride );
        m_pD3DDevice->SetIndices( m_pMesh[i]->m_pIndexBuffer );

        // Render the vertex buffer
        m_pD3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLESTRIP, 0, 0,
                                           BlockWidth * BlockHeight, 0,
                                           m_nPrimitiveCount );
    }
```

If the device supports single pass multi-texturing then the function can exit at this point. However, if single pass blending was not supported then only stage 0 would have been used when rendering the above terrain polygons. This means the color of the base texture in stage 0 would have been used as the final color for the terrain. We will have to now render the terrain again using the detail texture. The following code assigns the detail map to stage 0 and adjusts the texture coordinate index state of this stage so that it will use the second set of texture coordinates in each vertex. We also enable alpha blending and setup the frame buffer blending modes to get the proper blending effect.

```
    // If we are not using single pass we render detail in a second pass
    if ( !m_bSinglePass )
    {
        // Set our detail texture in stage 0, use 2nd texture coordinates.
        m_pD3DDevice->SetTexture( 0, m_pDetailTexture  );
        m_pD3DDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 1 );

        // Enable alpha blending to blend the dest and src colors together
        m_pD3DDevice->SetRenderState( D3DRS_SRCBLEND , D3DBLEND_DESTCOLOR );
        m_pD3DDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_SRCCOLOR );
        m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
```

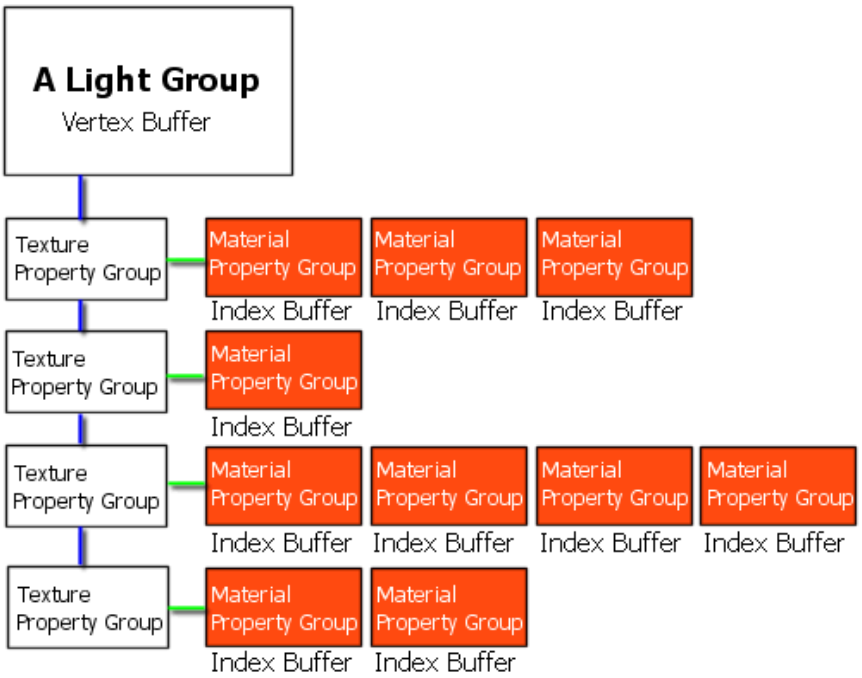
```
        // Render Each Mesh
        for ( i = 0; i < m_nMeshCount; i++ )
        {
            // Set the stream sources
            m_pD3DDevice->SetStreamSource(0, m_pMesh[i]->m_pVertexBuffer, 0,
                                          m_pMesh[i]->m_nStride );
            m_pD3DDevice->SetIndices( m_pMesh[i]->m_pIndexBuffer );

            // Render the vertex buffer
            m_pD3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLESTRIP, 0, 0,
                                               BlockWidth * BlockHeight, 0,
                                               m_nPrimitiveCount );
        }

        // Reset states for next call
        m_pD3DDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );
        m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );

    } // End if requires second pass
}
```

Remember to reset alpha blending to false and the texture coordinate index of stage 0 to zero for the next time we render.

> **Note:** Be sure to set your texture stage textures to NULL when you are finished rendering your scene. If you do not do this, memory leaks may occur.

Lab Project 6.3: Scene Texturing

Lab Project 6.3 implements a textured version of the IWF scene imported in the last lesson (see Lab Project 5.2). This project will demonstrate loading and using compressed textures, and blending vertex lighting with texture colors in texture stage 0. The demo will continue to use the DirectX lighting pipeline as well as the light group method described in detail in the last lesson. We will add textures to the property groups and assign each texture property group an array of child material property groups. These material property groups will store the triangles (their indices). A light group will continue to maintain the main vertex buffer used by its children. The current design is shown below.



We see in the above diagram that each light group will have a CPropertyGroup array. Each one will contain a texture index rather than a material index as we used in Chapter 5. Each texture CPropertyGroup will not contain any indices but they will contain an array of child CPropertyGroups. Each child contains an index into the main material array as well as the triangle indices. These indices describe the triangles that use the material property, the texture of the parent property group, and the lights of the top level light group. The result is that we are batching first by light group, then by texture, and finally by material. Rendering would then follow the approach:

- Begin Scene
- For Each Light Group (L)
- Setup Lights
- For Each Texture Group (T)
- SetTexture T->Texture
- For Each Material Group(M) of Texture Group (T)
- Set Material M->Material
- DrawIndexedPrimitive (L->Vertices , M->Indices)
- End Scene

Setting a texture is a more expensive operation than setting a material. This is especially true if the texture is not currently in video memory and has to be uploaded by the memory manager. This is why we make textures the primary key of the property chain. This final arrangement ensures that we are minimizing the number of SetLight, SetTexture, and SetMaterial calls and maximizing the number of primitives we can render with a single call to DrawIndexedPrimitive.

Note that we will only discuss code changes and additions in this chapter, so be sure that you are comfortable with the initial project in Chapter 5 before continuing.

This level does not use any multi-texturing so we will require only a single set of 2D texture coordinates per vertex. Our vertex class will also store a vertex normal for lighting purposes.

```
class CVertex
{
public:
   float       x;
   float       y;
   float       z;
   D3DXVECTOR3 Normal;
   float       tu;
   float       tv;
};
```

The corresponding flexible vertex flags are shown below.

```
#define VERTEX_FVF  D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1
```

The CGameApp Class

The CGameApp class has carried over all of the changes from Lab Project 6.2. It includes a TestDeviceCaps function which queries filters and related device capabilities. We have added support for texture format querying. The TestDeviceCaps function will first determine support for compressed textures. If that fails then it will try 32-bit textures and ultimately drop down to 16-bit textures if necessary. Once we find a supported texture format we will store this information in the members variables defined below.

D3DFORMAT m_TextureFormat;
This is the best texture format supported by the current device. The CScene class will use it in the D3DXCreateTextureFromFileEx function so that the images are stored in this format. By default, the TestDeviceCaps function will consider the compressed texture format D3DFMT_DXT1 optimal. We selected this format because the textures used in this project contain no alpha information.

D3DFORMAT m_AlphaFormat;
This variable will not actually be used in this project, but will be used in future projects. TestDeviceCaps will also try and find the best supported texture format that supports an alpha channel. We can use this format if we have texture images that include alpha channels and require an appropriate format. By default, the TestDeviceCaps function considers compressed alpha format D3DFMT_DXT3 optimal, but will fall back to uncompressed 32-bit alpha formats or even 16-bit alpha formats if necessary.

CGameApp::TestDeviceCaps

Below we show the last section of the CGameApp::TestDeviceCaps function. This code finds the best supported texture formats for both alpha and non-alpha texture surfaces. The rest of the function is unchanged from the previous incarnation.

```
ULONG      Ordinal = pSettings->AdapterOrdinal;
D3DDEVTYPE Type    = pSettings->DeviceType;
D3DFORMAT  AFormat = pSettings->DisplayMode.Format;

m_TextureFormat    = D3DFMT_UNKNOWN;
m_AlphaFormat      = D3DFMT_UNKNOWN;

// find texture formats we would like to use
// Prefer compressed textures in this demo
if(SUCCEEDED(m_pD3D->CheckDeviceFormat(Ordinal, Type, AFormat, 0,
                                       D3DRTYPE_TEXTURE, D3DFMT_DXT1)))
         m_TextureFormat = D3DFMT_DXT1;
else
if(SUCCEEDED(m_pD3D->CheckDeviceFormat(Ordinal, Type, AFormat, 0,
                                       D3DRTYPE_TEXTURE, D3DFMT_X8R8G8B8 )))
         m_TextureFormat = D3DFMT_X8R8G8B8;
else
if(SUCCEEDED(m_pD3D->CheckDeviceFormat(Ordinal, Type, AFormat, 0,
                                       D3DRTYPE_TEXTURE, D3DFMT_R5G6B5 )) )
         m_TextureFormat = D3DFMT_R5G6B5;
else
if(SUCCEEDED(m_pD3D->CheckDeviceFormat(Ordinal, Type, AFormat, 0,
                                       D3DRTYPE_TEXTURE, D3DFMT_X1R5G5B5 )) )
         m_TextureFormat = D3DFMT_X1R5G5B5;
```

After this code executes we have the best compatible non-alpha texture format supported by the device. We store it in the CGameApp::m_TextureFormat member so that it can be used later to create our textures. We now repeat the process for the best alpha capable texture format.

```
// Find alpha texture formats we would like to use
// Prefer compressed textures in this demo
if ( SUCCEEDED( m_pD3D->CheckDeviceFormat( Ordinal, Type, AFormat, 0,
                                           D3DRTYPE_TEXTURE, D3DFMT_DXT3)))
        m_AlphaFormat = D3DFMT_DXT3;

else

if ( SUCCEEDED( m_pD3D->CheckDeviceFormat( Ordinal, Type, AFormat, 0,
                                           D3DRTYPE_TEXTURE, D3DFMT_A8R8G8B8)))
        m_AlphaFormat = D3DFMT_A8R8G8B8;

else

if ( SUCCEEDED( m_pD3D->CheckDeviceFormat( Ordinal, Type, AFormat, 0,
                                           D3DRTYPE_TEXTURE, D3DFMT_A1R5G5B5)))
        m_AlphaFormat = D3DFMT_A1R5G5B5;

else

if ( SUCCEEDED( m_pD3D->CheckDeviceFormat( Ordinal, Type, AFormat, 0,
                                           D3DRTYPE_TEXTURE, D3DFMT_A4R4G4B4)))
        m_AlphaFormat = D3DFMT_A4R4G4B4;

   return true;
}
```

The last resort alpha format is the 16-bit ARGB4444 (supported by some hardware). This is a non-optimal format as it only has 4 bits per color component and thus can only store 15 shades of intensity. When converting from 32-bit pixel formats into 16-bit ARGB4444 format you will usually notice a distinct loss in image quality because of the limited number of colors it supports.

CGameApp::SetupRenderStates

The only other function that has changed in the CGameApp class from Lab Project 5.2 is CGameApp::SetupRenderStates -- called when the device is first created or reset. The function now calls TestDeviceCaps to gather information about supported texture formats, filter types, etc. Then it sets up the render states we have come to expect. We set an ambient light level, enable lighting, enable specular highlights, and the Z-Buffer.

```
void CGameApp::SetupRenderStates()
{
    // Test the device capabilities.
    if (!TestDeviceCaps( )) { PostQuitMessage(0); return; }

    // Setup our D3D Device initial states
    m_pD3DDevice->SetRenderState( D3DRS_ZENABLE, D3DZB_TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_DITHERENABLE,  TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_SHADEMODE, D3DSHADE_GOURAUD );
    m_pD3DDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CCW );
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_SPECULARENABLE, TRUE );
```

```
    m_pD3DDevice->SetRenderState( D3DRS_AMBIENT, 0x0D0D0D );
```

We use the same code from the previous terrain demo to set the filter types and anisotropy settings.

```
    // Set up sampler states.
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_MINFILTER    , m_MinFilter );
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_MAGFILTER    , m_MagFilter );
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_MIPFILTER    , m_MipFilter );
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_MAXANISOTROPY, m_Anisotropy );
```

We set the color operation in this blending stage to modulate the diffuse color calculated by the lighting pipeline with the texel color sampled from the texture in stage 0. The diffuse color is bound to argument 1 and the texture color to argument 2. We also inform the stage that it should use the first (and only) set of texture coordinates in our vertices.

```
    // Set texture / addressing / color ops
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLOROP  , D3DTOP_MODULATE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );
```

Next we set the fill mode and vertex format our application will be using and then set the view and projection matrices as usual.

```
    // Setup option dependant states
    m_pD3DDevice->SetRenderState( D3DRS_FILLMODE, m_FillMode );

    // Setup our vertex FVF code
    m_pD3DDevice->SetFVF( VERTEX_FVF );

    // Update our device with our camera details (Required on reset)
    m_pCamera->UpdateRenderView( m_pD3DDevice );
    m_pCamera->UpdateRenderProj( m_pD3DDevice );
```

Now we call the new CScene function SetTextureFormat to inform the class of the texture format that it should use for texture creation.

```
    // Inform texture loading objects which format to use
    m_Scene.SetTextureFormat( m_TextureFormat );
```

Finally, we call SelectMenuItems to enable/disable the supported filter and fill mode options that were recorded in the TestDeviceCaps function.

```
    // Set up the menu items (which may have changed during device validations)
    SelectMenuItems();
}
```

The CScene Class

Recall that level loading and light group and property group initialization begins in CScene::LoadScene. It is called from the CGameApp::BuildObjects function. Also recall that the CScene::Render function called by CGameApp::FrameAdvance actually renders the scene. Therefore, it is the CScene class handles most of our logic. Let us start by examining the changes we have had to make to this class in order to support texturing. This includes loading textures, building the light groups and taking textures into account, and batch rendering the scene with textures.

CScene includes a few new member variables to aid our texture management tasks. Below is a complete list of the CScene member variables with the new members in bold.

```
D3DMATERIAL9       *m_pMaterialList;          // Array of material structures
LPDIRECT3DTEXTURE9 *m_pTextureList;           // Array of texture pointers
D3DLIGHT9          *m_pLightList;             // Array of light structures
D3DLIGHT9           m_DynamicLight;           // Single dynamic light for testing
CLightGroup       **m_ppLightGroupList;       // Array of individual lighting groups
ULONG               m_nMaterialCount;         // Number of materials stored
ULONG               m_nTextureCount;          // Number of textures stored
ULONG               m_nLightCount;            // Number lights stored here
ULONG               m_nLightGroupCount;       // Number of light groups stored here
long                m_nWaterTexture;          // Index for our animating tex coords
D3DXMATRIX          m_mtxTexture;             // Texture matrix animates tex coords
ULONG               m_nReservedLights;        // Number of light slots to leave empty
ULONG               m_nLightLimit;            // Number of device lights available
LPDIRECT3DDEVICE9   m_pD3DDevice;             // Direct3D Device
bool                m_bHardwareTnL;           //  TnL support
D3DFORMAT           m_fmtTexture;             // Texture format for building textures
```

LPDIRECT3DTEXTURE9 *m_pTextureList;
This member points to an array of IDirect3DTexture9 interfaces -- one for each texture loaded from the IWF file. Each texture property group will contain an index into this array describing the texture the faces use.

ULONG m_nTextureCount;
The number of textures in the above texture array.

long m_nWaterTexture;
This member stores the index of the texture used as our water texture. During rendering of the scene, faces that use the water texture will have a texture transformation matrix enabled before being rendered. Each frame, we will adjust the translation vector in this matrix to animate their texture coordinates and thus make the water appear to flow using a simple scrolling animation.

D3DXMATRIX m_mtxTexture;
This is our texture matrix. Each frame we will increment the m31 element of the matrix to create the scrolling animation. The range is [0, 1] and we will loop back around at 1.0 to effect the scrolling. After we adjust this matrix, we call IDirect3DDevice9::SetTransform to send it to the device.

D3DFORMAT m_fmtTexture;
This contains the texture format we will use to load/create our textures. This was determined in the CGameApp::TestDeviceCaps function.

CScene::LoadScene

The LoadScene function contains only one new line. It is shown below in bold.

```
bool CScene::LoadScene(TCHAR *strFileName, ULONG LightLimit /* = 0 */,
                       ULONG LightReservedCount /* = 0 */ )
{
    CFileIWF File;

    // Attempt to load the file
    File.Load( strFileName );

    // Copy over the entities, textures  and materials we want from the file
    if (!ProcessEntities( File )) return false;
    if (!ProcessMaterials( File )) return false;
    if (!ProcessTextures( File )) return false;

    // Store values
    m_nLightLimit     = LightLimit;
    m_nReservedLights = LightReservedCount;

    // Check for unlimited light sources
    if ( m_nLightLimit == 0 )
         m_nLightLimit = m_nLightCount + LightReservedCount;

    // Now process the meshes and extract the required data
    if (!ProcessMeshes( File )) return false;

    // Build vertex / index buffers for each light group
    for ( USHORT i = 0; i < m_nLightGroupCount; i++ )
    {
       if(!m_ppLightGroupList[i]->BuildBuffers(m_pD3DDevice,
                                               m_bHardwareTnL,true ))
            return false;
    }

    // Allow file loader to release any active objects
    File.ClearObjects();

    return true;
}
```

Recall from Chapter 5 that ProcessEntities and ProcessMaterials extract the lights and materials from the IWF file and store them in the CScene::m_pLightList and m_pMaterialList arrays respectively. ProcessTextures is similarly used to extract the texture names from the IWF file, load the files, and add the textures to the CScene::m_pTextureList array. ProcessMeshes does the work of building the light groups and property groups and assigning vertices and indices.

CScene::ProcessTextures

The CFileIWF object (an IWF SDK class used to facilitate file loading) contains a vector of **TEXTURE_REF** structures for each texture used by the scene. The **TEXTURE_REF** structure is shown below as defined in LibIWF.h. In IWF files, texture images can be stored either as a list of file names, or the image data itself can be embedded inside the IWF file in an arbitrary format. GILES exports only the texture file names. Each **TEXTURE_REF** structure in the CFileIWF::m_vpTextureList vector describes how the texture is stored as well as the name of the file or the format of the internal image data, depending on how the file was created.

```
typedef struct _TEXTURE_REF
{
    UCHAR       TextureSource;
    USHORT      NameLength;
    char       *Name;
    UCHAR       TextureFormat;
    USHORT      TextureSize;
    UCHAR      *TextureData;
}TEXTURE_REF;
```

UCHAR TextureSource
This member indicates whether the texture is stored as image data or only as a file name. This can be one of two values shown below (defined in LibIWF.h):

```
#define TEXTURE_EXTERNAL                    0x1
#define TEXTURE_INTERNAL                    0x2
```

All of the textures in our IWF files will be of type **TEXTURE_EXTERNAL** in this demo, so we will load them ourselves using the stored file names and the D3DX functions discussed in the lesson.

USHORT NameLength
UCHAR *Name
If TextureSource is set to **TEXTURE_EXTERNAL** then Name will contain the texture file name and NameLength will contain the length of the file name in bytes.

UCHAR TextureFormat
If the **TextureSouce** member is set to **TEXTURE_INTERNAL** then this member identifies the format of the internal image data. It can be any one of the following values defined in LibIWF.h:

```
#define TEXFORMAT_RAW                               0x1
#define TEXFORMAT_CUSTOM                            0x2
#define TEXFORMAT_BMP                               0x3
#define TEXFORMAT_JPEG                              0x4
#define TEXFORMAT_TGA                               0x5
#define TEXFORMAT_PNG                               0x6
#define TEXFORMAT_PCX                               0x7
#define TEXFORMAT_GIF                               0x8
#define TEXFORMAT_PSD                               0x9
```

```
#define TEXFORMAT_TIFF                                  0xA
#define TEXFORMAT_PPM                                   0xB
```

There are many well known formats listed here in addition to a custom format. The latter is used if you wish to store the image data in your own application specific formats. This is not used when TextureSource is set to **TEXTURE_EXTERNAL**.

USHORT TextureSize
UCHAR * TextureData
If TextureSource is set to **TEXTURE_INTERNAL** then TextureData is a BYTE pointer to the image data and TextureSize describes the size of the image data in bytes.

Since GILES exports only texture file names, our ProcessTextures function will need to extract the file names from each **TEXTURE_REF** structure and then load the data. Every iwfSurface in the CFileIWF file also stores a texture index into the **TEXTURE_REF** vector describing the texture it uses. Therefore, we can loop through each **TEXTURE_REF** in the vector, extract the file name, load the texture and add it to our CScene::m_pTextureList array. Then, every iwfSurface can index into this array instead.

```
bool CScene::ProcessTextures( const CFileIWF& File )
{
    ULONG i;
    char  FileName[MAX_PATH];

    // Allocate enough room for all of our textures
    m_pTextureList = new LPDIRECT3DTEXTURE9[ File.m_vpTextureList.size() ];
    if ( !m_pTextureList ) return false;
    m_nTextureCount = File.m_vpTextureList.size();

    // Loop through and build our textures
    ZeroMemory( m_pTextureList, m_nTextureCount * sizeof(LPDIRECT3DTEXTURE9));
```

Here we exact the number of textures in the file and assign this to the CScene::m_nTextureCount variable. We also allocate the IDirect3DTexture9 pointer array (CScene::m_pTextureList) to hold this many IDirect3DTexture9 pointers. Now we can loop through each **TEXTURE_REF** stored in the CFileIWF::m_vpTextureList vector and extract the file name.

```
    for ( i = 0; i < File.m_vpTextureList.size(); i++ )
    {
        // Retrieve pointer to file texture
        TEXTURE_REF * pFileTexture = File.m_vpTextureList[i];

        // Skip if this is an internal texture (not supported by this demo)
        if ( pFileTexture->TextureSource != TEXTURE_EXTERNAL ) continue;

        // Build the final texture path
        strcpy( FileName, TexturePath );
        strcat( FileName, pFileTexture->Name );
```

Now that we have the file name we can use the D3DXCreateTextureFromFileEx function to load the image file and create the texture. Notice that we pass the CScene::m_fmtTexture format variable as the texture format we wish to use for our textures. Hopefully this will be the compressed texture format discussed earlier. The final texture is loaded into the relevant slot in the texture pointer array.

```
        // Load the texture from file
        D3DXCreateTextureFromFileEx(m_pD3DDevice, FileName, D3DX_DEFAULT, D3DX_DEFAULT,
                                    D3DX_DEFAULT, 0, m_fmtTexture, D3DPOOL_MANAGED,
                                    D3DX_DEFAULT, D3DX_DEFAULT, 0, NULL, NULL,
                                    &m_pTextureList[i] );
```

Finally, we check the name of the texture we are loading to see if it has the same name as the water texture. If it does, then we record the index of this texture in the CScene::WaterTexture member variable. We use this during rendering to enable the texture transformation matrix for faces that use this texture.

```
     // Store the index to the texture we want to animate if this is the one
     if(strcmp(pFileTexture->Name, "Water Bump Map 001.jpg") == 0)
                                m_nWaterTexture = i;

   } // Next Texture

   // Success!
   return true;
}
```

When this function returns, we have all of our textures loaded with their pointers in our texture array. The ProcessEntities and ProcessMaterials functions are unchanged in this demo and simply load the materials and lights into the CScene arrays.

CScene::ProcessMeshes

The ProcessMeshes function is responsible for building the light groups and sorting the scene faces into the relevant light groups, texture property groups and material property groups. While much of the function remains the same as the version in Lab Project 5.2, some of the order in which tasks are executed has changed to support texture property groups.

We will modify the BuildLightGroups function and call it only once at the start of the ProcessMeshes function. It works in almost exactly the same way -- it loops through every iwfSurface stored in the file and calculates the light group it belongs to or creates a new one if a match could not be found. However we are now looping through every face in the file and not just the surfaces in the material vector as was the case before. Once we find a matching light group, we will simply store the light group index inside the iwfSurface. It just so happens that the iwfSurface structure includes a member variable that we are not using called iwfSurface::CustomData, so we will use it to store this index. This means that the BuildLightGroups function has had all of the code that searches for the correct property groups removed and is therefore simplified significantly. When BuildLightGroups returns control back

to the ProcessMeshes function, the CScene::m_mppLightGroupList will contain all of the light groups created along with their array of lights, but it will contain no face data or child property groups at this point. The iwfSurfaces stored inside the CFileIWF object however will each contain the light group to which they belong. ProcessMeshes will then assign each surface to the light group it belongs to and create the appropriate property groups. We will not look at the BuildLightGroups function since it is virtually identical to its prior version.

```
bool CScene::ProcessMeshes( CFileIWF & pFile )
{
    long i, j, k, l, m, TextureIndex, MaterialIndex;
    CLightGroup    * pLightGroup = NULL;
    CPropertyGroup * pTexProperty = NULL;
    CPropertyGroup * pMatProperty = NULL;

    // Allocate the light groups, and assign the surfaces to them
    if (!BuildLightGroups( pFile )) return false;
```

The first thing this function does is call BuildLightGroups. When this function returns the CScene::m_ppLightGroupList array contains pointers to all the light groups the scene will need. The light groups will have their lights assigned to them but not yet have any surfaces. The iwfSurfaces stored inside the CFileIWF object will each contain an index describing which light group they belong to.

Now we enter a series of nested loops to iterate through the textures, then the materials, followed by the meshes, and finally the individual surfaces. We want to find every surface that uses the current texture, and inside that loop find the surfaces that also use each material.

```
    // For each texture (including no texture, -1)
    for ( l = -1; l < (signed)m_nTextureCount; l++ )
    {
        // For each material (including no material, -1)
        for ( m = -1; m < (signed)m_nMaterialCount; m++ )
        {
            // Loop through each mesh
            for ( i = 0; i < pFile.m_vpMeshList.size(); i++ )
            {
                iwfMesh * pMesh = pFile.m_vpMeshList[i];

                // and then loop through each surface in this mesh
                for ( j = 0; j < pMesh->SurfaceCount; j++ )
                {
                    iwfSurface * pSurface = pMesh->Surfaces[j];
```

The surface stores both the material and texture indices it is using so we retrieve both of these values:

```
                    // Determine the material and texture indices we are using.
                    MaterialIndex = -1;
                    TextureIndex  = -1;

// Get material Index for this face
if((pSurface->Components & SCOMPONENT_MATERIALS) && pSurface->ChannelCount > 0)
   MaterialIndex = pSurface->MaterialIndices[0];
```

```
// Get Texture Index for this face
if((pSurface->Components & SCOMPONENT_TEXTURES) && pSurface->ChannelCount > 0)
   TextureIndex  = pSurface->TextureIndices[0];
```

We are looking for a surface that uses the current texture and the current material. If this surface does not then skip it so that it can be processed in future iterations.

```
                    // Skip if this is not in order
                    if ( TextureIndex != l || MaterialIndex != m ) continue;
```

Now we need to find the light group the current surface belongs to. Recall that the BuildLightGroups function stored the light group index in the iwfSurface::CustomData member.

```
                    // Retrieve the lightgroup pointer for this surface
                    pLightGroup = (CLightGroup*)pSurface->CustomData;
```

We need to search the current light group child property groups to see if one already exists that uses this texture. If there is no property group in the light group which contains the current texture then we create a new one. Unlike the earlier version of this demo that used no textures, each child property group will now have its m_nPropertyData member used to store a texture index instead of a material index. Therefore, we search for property groups in the current light group using this search key as shown below.

```
                   // see if we already have a property group for this texture
                   for ( k = 0; k < pLightGroup->m_nPropertyGroupCount; k++ )
                   {
                      // Break if texture index matches
                      if((long)pLightGroup->m_pPropertyGroup[k]->m_nPropertyData
                          == TextureIndex )
                          break;
                   }
```

If a property group could not be found then we add a new one and store the current texture index in it. Notice that we now use the PROPERTY_TEXTURE enumerated type to describe the property group as a texture property group:

```
                    // If we didn't have this property group, add it
                    if ( k == pLightGroup->m_nPropertyGroupCount )
                    {
                        if ( pLightGroup->AddPropertyGroup( ) < 0 )
                           return false;

                        // Set up property group data for primary key
                        pTexProperty = pLightGroup->m_pPropertyGroup[ k ];
                        pTexProperty->m_PropertyType  = CPropertyGroup::PROPERTY_TEXTURE;
                        pTexProperty->m_nPropertyData = (ULONG)TextureIndex;
                    }

                    // Process for secondary key (material)
                    pTexProperty = pLightGroup->m_pPropertyGroup[ k ];
```

At this point, we have a pointer to the property group that uses the texture. Unlike the earlier version of this demo, we will use the CPropertyGroup::m_pPropertyGroup array so that each property group can store pointers to child property groups. So we have the light group the surface belongs to and the property group the texture belongs to. We now have to find a child property group for the texture property group that matches the current material for this surface. For all child property groups of texture property groups, the CPropertyGroup::m_nPropertyData member will hold a material index. If we do not find a child property group that contains the material index of the current surface, we create a new material group and add it to the texture property groups array.

```
                    // see if we already have a property group for this material
                    for ( k = 0; k < pTexProperty->m_nPropertyGroupCount; k++ )
                    {
                        // Break if texture index matches
                        if((long)pTexProperty->m_pPropertyGroup[k]->m_nPropertyData
                           == MaterialIndex )
                          break;
                    }

                    // If we didn't have this property group, add it
                    if ( k == pTexProperty->m_nPropertyGroupCount )
                    {
                        if ( pTexProperty->AddPropertyGroup( ) < 0 )
                           return false;

                         // Set up property group data for primary key
                         pMatProperty = pTexProperty->m_pPropertyGroup[ k ];
                         pMatProperty->m_PropertyType  =  \
                                              CPropertyGroup::PROPERTY_MATERIAL;
                         pMatProperty->m_nPropertyData = (ULONG)MaterialIndex;
                         pMatProperty->m_nVertexStart  = \
                                                 pLightGroup->m_nVertexCount;
                         pMatProperty->m_nVertexCount  = 0;
                     }
```

We now have the light group, the texture group inside the light group, and the material group inside the texture group that the surface belongs to. Therefore, we can call the ProcessIndices and ProcessVertices functions (unchanged from last demo) to add the vertices of the surface to the light group and the indices of the surface to the material group.

```
                     // Process the vertices / indices
                     pMatProperty = pTexProperty->m_pPropertyGroup[ k ];
                     if(!ProcessIndices( pLightGroup, pMatProperty, pSurface ) )
                          return false;
                     if(!ProcessVertices( pLightGroup, pMatProperty, pSurface) )
                          return false;

                } // Next Surface

            } // Next Mesh

        } // Next Material

    } // Next Texture
```

We used the iwfSurface::CustomData member to hold the index of the texture temporarily. This member is usually used to store a pointer to custom data and as such when the CFileIWF object is deleted, this member is assumed to be a pointer and is deleted. If we do not clean up after ourselves, then the class would interpret the texture index as a custom data chunk address and try to delete it. This would not be good. So we will set them all to NULL again after we have finished using them.

```
    // Clear the custom data pointer so that it isn't released
    for ( i = 0; i < pFile.m_vpMeshList.size(); i++ )
    {
        iwfMesh * pMesh = pFile.m_vpMeshList[i];
        for ( j = 0; j < pMesh->SurfaceCount; j++ )
            pMesh->Surfaces[j]->CustomData = NULL;
    }

    // Success!!
    return true;
}
```

CScene::AnimateObjects

The CGameApp::FrameAdvance function calls CScene::AnimateObjects once per frame. This is a new function whose purpose is to increment the translation vector U coordinate in the texture matrix to scroll the water texture.

```
void CScene::AnimateObjects( CTimer & Timer )
{
    // Shift the texture coordinates along the U axis
    m_mtxTexture._31 += 0.5f * Timer.GetTimeElapsed();
    if ( m_mtxTexture._31 > 1.0f ) m_mtxTexture._31 -= 1.0f;
}
```

We scroll the U coordinate at a rate of 0.5 units per-second and wrap back to 0.0 when it exceeds 1.0. This matrix will be set in the CScene::Render function when rendering faces that have the water texture applied to them.

CScene::Render

This function has been changed to set the texture matrix and to render our scene using the new batching scheme.

```
void CScene::Render( )
{
    ULONG         i, j, k;
    CLightGroup * pLightGroup = NULL;
    ULONG       * pLightList  = NULL;
```

First we loop through each light group and enable the specified lights.

```
    // Loop through each light group
    for ( i = 0; i < m_nLightGroupCount; i++ )
    {
        // Set active lights
        pLightGroup = m_ppLightGroupList[i];
        pLightList  = pLightGroup->m_pLightList;
        for ( j = m_nReservedLights; j < m_nLightLimit; j++ )
        {
            if ( (j - m_nReservedLights) >= (pLightGroup->m_nLightCount ) )
            {
                // Disable any light sources which should not be active
                m_pD3DDevice->LightEnable( j, FALSE );

            } // End if no more lights
            else
            {
                // Set this light as active
                m_pD3DDevice->SetLight( j,
                                        &m_pLightList[pLightList[j - m_nReservedLights]]);
                m_pD3DDevice->LightEnable( j, TRUE );
            } // End if set lights
        } // Next Light
```

Now that we have set the current light group, we set the light group vertex buffer as the current vertex stream for rendering.

```
        // Set vertex stream
        m_pD3DDevice->SetStreamSource(0, pLightGroup->m_pVertexBuffer, 0, sizeof(CVertex));
```

Now we loop through each of the texture property groups and set the current texture.

```
        // Now loop through and render the associated property groups
        for ( j = 0; j < pLightGroup->m_nPropertyGroupCount; ++j )
        {
            CPropertyGroup * pTexProperty = pLightGroup->m_pPropertyGroup[j];
            long TextureIndex = (long)pTexProperty->m_nPropertyData;

            // Set Properties
            if ( TextureIndex >= 0 )
            {
                m_pD3DDevice->SetTexture( 0, m_pTextureList[ TextureIndex ] );
            }
            else
            {
                m_pD3DDevice->SetTexture( 0, NULL );
            }
```

Note that we set the texture unless this property group has no texture applied to it. If any faces exist which do not have textures applied, they will be in a property group with a –1 texture index.

Next we check to see if this texture property group contains the water texture and if so, we set the texture coordinate transformation matrix in stage 0. We also inform the device that we are using 2D texture coordinates..

```
            // Set the texture matrix for our animating water example
            if ( TextureIndex == m_nWaterTexture )
            {
                m_pD3DDevice->SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS,
                                                   D3DTTFF_COUNT2 );
                m_pD3DDevice->SetTransform( D3DTS_TEXTURE0, &m_mtxTexture );
            }
```

For every material property group we set the material and render the triangles stored in that group.

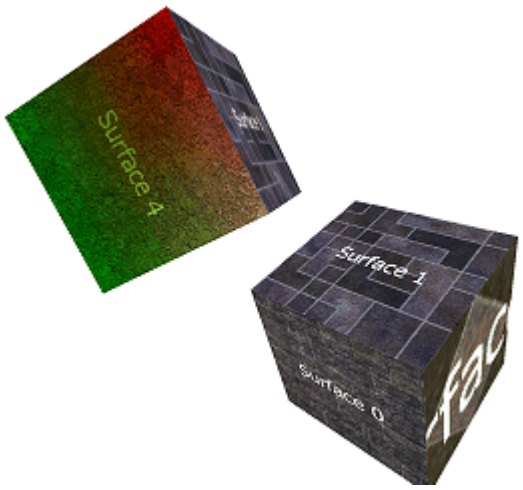
```
   // Render child property group
   for ( k = 0; k < pTexProperty->m_nPropertyGroupCount; ++k )
   {
      CPropertyGroup * pMatProperty = pTexProperty->m_pPropertyGroup[k];

      m_pD3DDevice->SetMaterial(&m_pMaterialList[(long)pMatProperty->m_nPropertyData]);
      m_pD3DDevice->SetIndices( pMatProperty->m_pIndexBuffer );
      m_pD3DDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,
                                          pMatProperty->m_nVertexStart , 0,
                                          pMatProperty->m_nVertexCount, 0,
                                          pMatProperty->m_nIndexCount / 3 );
   }
```

Finally, if we just rendered a water texture property group then we should remember to disable texture coordinate transformations.

```
            // Disable the texture matrix
            if ( TextureIndex == m_nWaterTexture )
            {
                m_pD3DDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS,
                                                    D3DTTFF_DISABLE );
            } // End if Water Texture

        } // Next Property Group

    } // Next Light Group

}
```

Lab Project 6.4 Writing to Surface with GDI

This project demonstrates locking top level texture surfaces and writing to them using GDI. Once we have written the text to each top level texture surface, we call the D3DXFilterTexture function to filter the text onto each MIP surface of the texture. As with Lab Project 6.1, the cube on the left has filtering techniques applied and the one on the right does not. This is actually a good demonstration of filtering results. We can see clearly how text becomes aliased as you zoom the camera out.



This demo is so similar to Lab Project 6.1 that we will only need to examine the section of the CGameApp::BuildObjects function that has changed. This function is called at application startup to build the cube meshes and to load the textures used by each face.

The following code snippet from CGameApp::BuildObjects is executed just after the cube mesh has been created. It loops through each of the six textures used by the application, retrieves an IDirect3DSurface9 interface to the top level MIP surface, and then uses this surface interface to get a device context. We then create a font and write the surface name on the image surface as a text string. Finally, the code calls D3DXFilterTexture to filter the changes down through the MIP chain.

```
// Load all 6 textures used in this example.
hRet = D3DXCreateTextureFromFile(m_pD3DDevice,"Data\\texture_01.jpg",&m_pTextures[0] );
hRet = D3DXCreateTextureFromFile(m_pD3DDevice,"Data\\texture_02.jpg",&m_pTextures[1] );
hRet = D3DXCreateTextureFromFile(m_pD3DDevice,"Data\\texture_03.jpg",&m_pTextures[2] );
hRet = D3DXCreateTextureFromFile(m_pD3DDevice,"Data\\texture_04.jpg",&m_pTextures[3] );
hRet = D3DXCreateTextureFromFile(m_pD3DDevice,"Data\\texture_05.jpg",&m_pTextures[4] );
hRet = D3DXCreateTextureFromFile(m_pD3DDevice,"Data\\texture_06.jpg",&m_pTextures[5] );

if ( FAILED(hRet) ) return false;

HDC   hDC   = NULL;
HFONT hFont = NULL, hOldFont = NULL;
LPDIRECT3DSURFACE9 pSurface = NULL;
char            Buffer[20];
LOGFONT                             logFont;
D3DSURFACE_DESC Desc;
RECT            rc;

// Set up common font settings
ZeroMemory( &logFont, sizeof(LOGFONT) );
_tcscpy( logFont.lfFaceName, "Tahoma" );

// Let's go crazy and draw on all of our textures
for ( i = 0; i < 6; i++ )
{
    // Skip if this texture failed to load
    if ( !m_pTextures[i] ) continue;
```

```
    // Retrieve this texture's top level surface and it's description
    if ( FAILED(m_pTextures[i]->GetSurfaceLevel(0, &pSurface )) ) continue;
    pSurface->GetDesc( &Desc );

    // Retrieve a device context for this surface
    if ( FAILED(pSurface->GetDC( &hDC )) ){ pSurface->Release(); continue; }

    // Get the actual height of this font, from the point size, in the DC
    logFont.lfHeight = -MulDiv(Desc.Width / 10, ::GetDeviceCaps(hDC, LOGPIXELSY), 72);

    // Create the actual Font Handle and select it
    hFont = ::CreateFontIndirect( &logFont );
    hOldFont = (HFONT)::SelectObject( hDC, hFont );

    // Set up our GDI rendering properties
    ::SetBkMode( hDC, TRANSPARENT );
    ::SetTextColor( hDC, 0xFFFFFF );

    // Set up the drawing rectangle from the surface description
    rc.left = 0; rc.right  = Desc.Width;
    rc.top  = 0; rc.bottom = Desc.Height;

    // Build a string and draw the text
    sprintf( Buffer, "Surface %i", i );
    ::DrawText(hDC, Buffer, strlen(Buffer), &rc,
               DT_CENTER | DT_SINGLELINE | DT_VCENTER);

    // Clean up DC (very important)
    ::SelectObject( hDC, hOldFont );
    ::DeleteObject( hFont );

    // Release the DC and the surface
    pSurface->ReleaseDC( hDC );
    pSurface->Release();

    // Filter the changes made to the top level surface, down to the mip-maps
    D3DXFilterTexture( m_pTextures[i], NULL, 0, D3DX_DEFAULT );

} // Next Texture
```

Note that we are not limited to getting an IDirect3DSurface9 interface to a texture surface only. We could alternatively use a surface interface (and the GetDC method) to draw to the frame buffer or even the depth buffer (although that will not be a very common undertaking).

Lab Project 6.5 Title Screen Demo

This final project loads an image from a file into a surface and copies the image surface to the frame buffer image using the IDirect3DDevice9::StretchRect function. This means that we can resize the window to any arbitrary size and the image will still fill the client area of the window.

This demo introduces some of the D3DX helper functions for working with surfaces as we will see now when we examine the code.

CGameApp::BuildObjects

In the CGameApp::BuildObjects function -- called at application start up and device reset -- we call D3DXGetImageInfoFromFile to open up the image file on disk and extract information about its dimensions and colour depth. We use this information to create an IDirect3DSurface9 object of the correct dimensions to load the image data into the surface. Here is the function in its entirety.

```
bool CGameApp::BuildObjects()
{
    HRESULT         hRet;
    D3DXIMAGE_INFO  Info;

    // Release previous objects just in case
    ReleaseObjects();

    // Retrieve device settings
    CD3DSettings::Settings * pSettings = m_D3DSettings.GetSettings();

    // Get the source file info
    if ( FAILED(D3DXGetImageInfoFromFile( "Data\\Image.jpg", &Info ))) return false;

    // Create the off screen surface
    hRet = m_pD3DDevice->CreateOffscreenPlainSurface(Info.Width, Info.Height,
                                                     pSettings->BackBufferFormat,
                                                     D3DPOOL_DEFAULT, &m_pSurface, NULL);
    if ( FAILED(hRet) ) return false;

    // Load in the image
    hRet = D3DXLoadSurfaceFromFile( m_pSurface, NULL, NULL, "Data\\Image.jpg", NULL,
                                    D3DX_DEFAULT, 0, NULL );
    if ( FAILED(hRet) ) return false;

    // Success!
    return true;
}
```

CGameApp::FrameAdvance

Now that the image has been loaded into our surface, all that is left to do is blit it to the frame buffer in our render loop. Admittedly we could get away with just copying the data once, but we put it in our render loop so that it is copied every frame in case we wish to render some 3D graphics over the top of it. This might be useful if you wanted to use the image as a backdrop for a 3D scene.

This function retrieves a pointer to an IDirect3DSurface9 interface for the frame buffer and calls the IDirect3DDevice9::StretchRect function to copy the offscreen surface image.

```
// Begin Scene Rendering
m_pD3DDevice->BeginScene();

LPDIRECT3DSURFACE9 pBackBuffer = NULL;
if(SUCCEEDED(m_pD3DDevice->GetBackBuffer( 0, 0, D3DBACKBUFFER_TYPE_MONO, &pBackBuffer )))
{
      if(m_pSurface)
        m_pD3DDevice->StretchRect( m_pSurface, NULL, pBackBuffer, NULL, D3DTEXF_NONE );

      pBackBuffer->Release();
}

// End Scene Rendering
m_pD3DDevice->EndScene();

// Present the buffer
if ( FAILED(m_pD3DDevice->Present( NULL, NULL, NULL, NULL )) ) m_bLostDevice = true;
```

Review Questions

1. What is MIP mapping and how does it improve visual quality?
2. Do MIP maps consume more memory than regular textures?
3. Are there are speed benefits to using MIP maps?
4. What is Bilinear filtering?
5. What is Trilinear filtering?
6. What is Anisotropic filtering?
7. What are texture coordinates?
8. Where do we store texture coordinates?
9. What is the pitch of a surface and how do we use it?
10. What is a texel?
11. Why are compressed texture formats beneficial?
12. How does the `D3DTADDRESS_CLAMP` texture addressing mode work?
13. What is a detail texture?
14. How many texture stages would we need to blend three textures onto a polygon in a single pass?
15. Can we use texturing and lighting together?
16. Are texture sizes limited to 256x256?
17. What does the magnification filter do?
18. What does the minification filter do?
19. What does the MIP filter do?
20. Can you lock textures created in the D3DPOOL_DEFAULT resource pool?
21. Can we use D3DPOOL_MANAGED surfaces as source surface parameters to the StretchRect and UpdateSurface functions?

Appendix A Texture Stage, Render, and Sampler States

New Render States Table

Render State	Parameters	Description
D3DRS_ALPHABLENDENABLE	True or False	Enables alpha blending with the frame buffer. The fragment color output from the texture stages is involved in a blending operation based on source and destination blend modes that we specify.
D3DRS_SRCBLEND	A member of the **D3DBLEND** eumerated type.	When alpha blending is enabled this state specifies how the source color is to be blended with the frame buffer.
D3DRS_DESTBLEND	A member of the **D3DBLEND** enumerated type.	When alpha blending is enabled this state specifies how the destination color is blended with the source color.
D3DRS_TEXTUREFACTOR	A **D3DCOLOR** value in the form 0xAARRGGBB. The default state is opaque white (0xFFFFFFFF)	This state can be used to set a constant color that can be accessed by the texture stage states during color and alpha blending in a texture stage. If a texture stage input argument is set to **D3DTA_TFACTOR**, this color will be used. If the state is blending two colors using the **D3DTOP_BLENDFACTORALPHA** color operation, the alpha component of this color is used to blend the two input colors.
D3DRS_WRAP0 To D3DRS_WRAP15	**D3DWRAPCOORD0** To **D3DWRAPCOORD3**	This render state allows us to set the wrapping mode used by a texture stage for a given axis. The wrapping mode is not to be confused with the texture addressing mode of a texture stage. The wrapping mode controls how the sampler interpolates between two sets of coordinates on a texture.
D3DRS_BLENDOP	A member of the **D3DBLENDOP** enumerated type.	This is used to set the calculation carried out when blending is enabled. The default operation is D3DBLENDOP_ADD as shown below: Src*SrcBlend + Dest*DestBlend

D3DRS_BLENDFACTOR	A **D3DCOLOR** value. The default is white 0xFFFFFFFF.	This allows us to set a constant color value that can be used when blending is enabled. Either the SourceBlend or DestinationBlend modes can be set to **D3DBLEND_BLENDFACTOR** to use this color in the blending operation.

New Texture Stage States Table

Texture Stage State	Parameters	Description
D3DTSS_COLOROP	A member of the **D3DTEXTUREOP** enumareted type	This allows us to choose a blending function for the color inputs to the texture stage. The default color operation for texture stage 0 is **D3DTOP_MODULATE** and for all other stages the default is **D3DTOP_DISABLE**.
D3DTSS_COLORARG1	One of the **D3DTA** constant values should passed	This is used to set the first color argument of a texture stage. The default is **D3DTA_TEXTURE** meaning that the first input color to the texture stage is the color sampled from the texture.
D3DTSS_COLORARG2	One of the **D3DTA** constant values should passed	This is used to set the second color argument of a texture stage. The default is **D3DTA_CURRENT** meaning that the second input color to the texture stage is the color output from a previous stage or the diffuse vertex color if this is used in stage 0..
D3DTSS_COLORARG0	One of the **D3DTA** constant values should passed	This is used to set the third color argument used in triadic color blending operations.
D3DTSS_ALPHAOP	A member of the **D3DTEXTUREOP** enumareted type	This allows us to choose a blending function for the alpha inputs to the texture stage. The default alpha operation for texture stage 0 is **D3DTOP_SELECTARG1**

		meaning the 1st alpha input is passed from the stage unaltered and no blending of alpha values takes place within the stage. The default for all other stages is **D3DTOP_DISABLE**.
D3DTSS_ALPHAARG1	One of the **D3DTA** constant values should passed	This is used to set the first alpha argument of a texture stage alpha pipeline. The default is **D3DTA_TEXTURE** meaning that the first alpha input to the texture stage is the alpha sampled from the texture or 0xFF if not alpha channel is present in the texture bound to this stage.
D3DTSS_ALPHAARG2	One of the **D3DTA** constant values should passed	This is used to set the second alpha argument of the current texture stage. The default is **D3DTA_CURRENT** meaning that the second alpha input to the texture stage is the alpha value output from a previous stage or the diffuse vertex alpha if this is used in stage 0.
D3DTSS_ALPHAARG0	One of the **D3DTA** constant values should passed	This is used to set the third alpha argument used in triadic alpha blending operations.
D3DTSS_TEXCOORDINDEX	Zero based index of the texture coordinate set within the vertices that this stage should use for sampling the texture.	The default values for each stage is for each stage to use the texture coordinate set that is equal to the stage number.For example, stage 0 uses the 1st set of texture coordinates in the vertex, stage 1 uses the 2nd, stage 2 uses the 3rd, and so on.
D3DTSS_TEXTURETRANSFORMFLAGS	A member of the **D3DTEXTURETRANSFORMFLAGS** enumerated type.	This is used to enable the texture coordinate transfomation matrix for a given stage. It also informs the renderer how many texture coordinates it should expect to be output from the matrix. In our example we are using 2D texture coordinate that we want passed to

		the renderer so we set this to **D3DTTFF_COUNT2**.
D3DTSS_CONSTANT	A **D3DCOLOR** that can be set as a per-stage constant color for use in color/alpha blending operations within the stage.	If we set any of the texture stages color or alpha arguments to **D3DTA_CONSTANT**, this color will be used as the color for that argument. Each stage can have its own contant color, unlike the **D3DRS_TEXTUREFACTOR** render state which is accessible from all texture stages.

New Sampler States Table

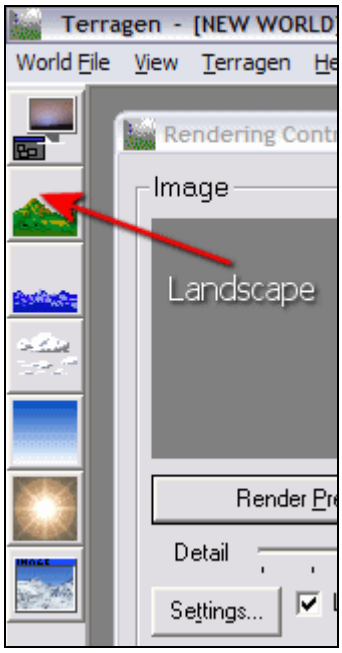
Sampler State	Parameters	Description
D3DSAMP_ADDRESSU	A member of the **D3DTEXTUREADDRESS** enumerated type.	Allows you to set the addressing mode used when U component of a texture coordinate is outside the 0.0 to 1.0 range. The default value for each stage is **D3DTADDRESS_WRAP** which means the texture is repeated along the U axis.
D3DSAMP_ADDRESSV	A member of the **D3DTEXTUREADDRESS** enumerated type.	Allows you to set the addressing mode used when the V component of a texture coordinate is outside the 0.0 to 1.0 range. The default value for each stage is **D3DTADDRESS_WRAP** which means the texture is repeated along the V axis.
D3DSAMP_ADDRESSW	A member of the **D3DTEXTUREADDRESS** enumerated type.	Allows you to set the addressing mode used when W component of a 3D texture coordinate is outside the 0.0 to 1.0 range. The default value for each stage is **D3DTADDRESS_WRAP** which means the texture is repeated

		along the W axis.
D3DSAMP_BORDERCOLOR	A **D3DCOLOR** specifying the border color to be used.	If any of the addressing modes are set to **D3DTSSADDRESS_BORDER** for a given axis, then area of the polygon outside the 0.0 to 1.0 range are set to this border color. The default value is black 0x00000000.
D3DSAMP_MINFILTER	A member of the **D3DTEXTUREFILTERTYPE** enumerated type.	This sets the minification filter used when sampling texels from a texture bound to the stage.The default state for each stage is **D3DTEXF_POINT**.
D3DSAMP_MAGFILTER	A member of the **D3DTEXTUREFILTERTYPE** enumerated type.	This sets the magnification filter used when sampling texels from a texture bound to the stage.The default state for each stage is **D3DTEXF_POINT**.
D3DSAMP_MIPFILTER	A member of the **D3DTEXTUREFILTERTYPE** enumerated type.	This sets the MIP filter used when sampling texels from a texture bound to the stage that has MIP level surfaces.The default state for each stage is **D3DTEXF_NONE** which means the nearest MIP level is used.
D3DSAMP_MIPMAPLODBIAS	A value between -n and n	This allows us to bias the calculation used by the sampler when determining which MIP level is closest to the ideal MIP level. Effectively, this allows us to alter the point at which a new MIP level is selected based on distance. You will usually leave this at its default value of zero to allow Direct3D to select the correct MIP level for a given polygon without modification. If you feel that the MIP levels are being selected too late or too early you can adjust the selection

		formula by forcing the next MIP level to be selected sooner or later by specifying negative or positive values respectively.
D3DSAMP_MAXMIPLEVEL	A value between 0 and n-1 where n is the number of MIP levels present in a texture.	This allows us to clamp the largest MIP level used during rendering to the specified value. The default value is zero meaning that all textures in the MIP chain are used. If you set this value to 2 for example, then the largest MIP levels (0 and 1) would not be used when rendering.
D3DSAMP_MAXANISOTROPY	A DWORD describing the maximum anisotropic filtering level that should be used during texture sampling for a given stage.	The default value is 1 which is the quickest but least effective anisotropic filtering level. You can determine the maximum level of anisotropy available for a given device by checking the **`D3DCAPS9::MaxAnisotropy`**

Appendix B Making Terrain Textures in Terragen™

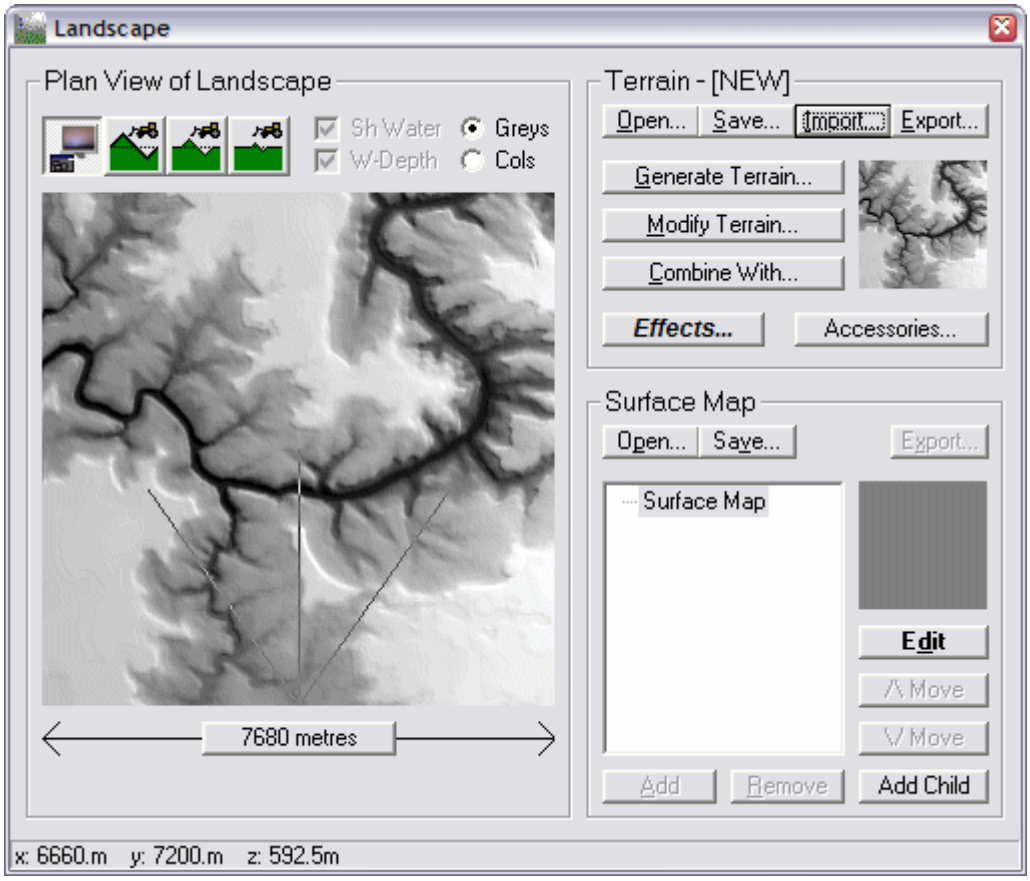
First we will need to generate or import a height-map. To do this, open the Landscape dialog. If this is not currently visible at start-up (or you have shut it down) this can be accessed by selecting the following toolbar button:



At this point, you will be presented with the main landscape dialog which allows us to generate / load terrain height-maps, set surface properties, and specify other landscape related effects. You can generate your own height-map here by selecting the Generate Terrain button. Simply provide the required properties in the newly opened dialog.

To import an existing terrain, select the Import button in the upper right portion of the Landscape dialog. Then select the RAW height-map file. We will use a 257x257 resolution, and store 8 bits per pixel (grayscale).

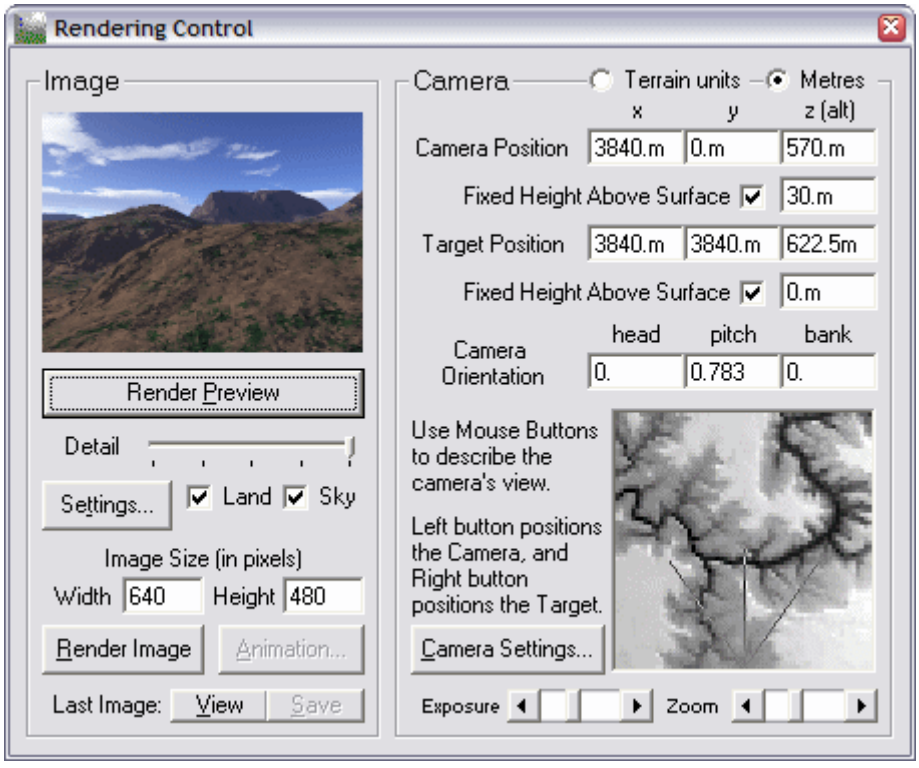
Once you have generated or imported your height-map data, the viewport on the left hand side should have been updated to reflect this as shown below:



You can now modify the terrain using several sculpting techniques. Either click the Modify Terrain button or use the sculpting tools with the Terrain & Bulldozer icons above the height-map view. You can paint directly onto the height-map view directly below them. If you have edited or generated a new terrain, you can export the file to disk using the Export button. This file will be saved as a 257x257 RAW file using 8 bits per pixel. You can then import this file into the demo applications we built in this course.

Now that we have our height-map imported, we need to give it some texture. Terragen comes with some very nice pre-created texture sets. These are called Surface Maps. They contain both the texture data itself and the parameters used to inform the application about where these materials should be applied (ex. only on sheer cliff faces). This is referred to as the landscape's Ecology. To provide these settings, we will need to open a surface map. Just select the Open button inside the Surface Map group on the Landscape dialog shown above. In this example, we will choose one of the pre-packaged surface maps called DesertAndGrass.srf. After you have opened the map, the list containing the words Surface Map should be replaced with a list of all map property types that will be used during the generation of the scene. We will leave these as default settings for now, but you can modify their properties easily. For example, try selecting Sand, and press then the Edit button seen on the right hand side.

We are done with the Landscape dialog for now, so you can either close it or minimize it. Next we need to open up the Rendering Control dialog by selecting the following toolbar button:



Your dialog should be similar to that shown to the left. Using the default settings (with the exception of the Detail slider which has been moved all the way to the right), and then selecting Render Preview will render a small image so that we can take a quick look at how the Terrain is currently shaping up.

At this point, we can see our height-map in the lower right portion of the dialog and the resulting image in the top left. We now need to set up our camera properties to ensure that we render from a top-down viewpoint.

The following list of steps will set up Terragen to render our terrain texture from the correct viewpoint:

- Un-check the Sky check box on the left hand side of the dialog.
- At the top of the Camera group, select the Terrain Units radio button.
- Un-check both Fixed Height Above Surface check boxes.
- Set Camera Position XYZ to (128.5, 128.5, 5500).
- Set Target Position XYZ to (128.5, 128.5, 0).

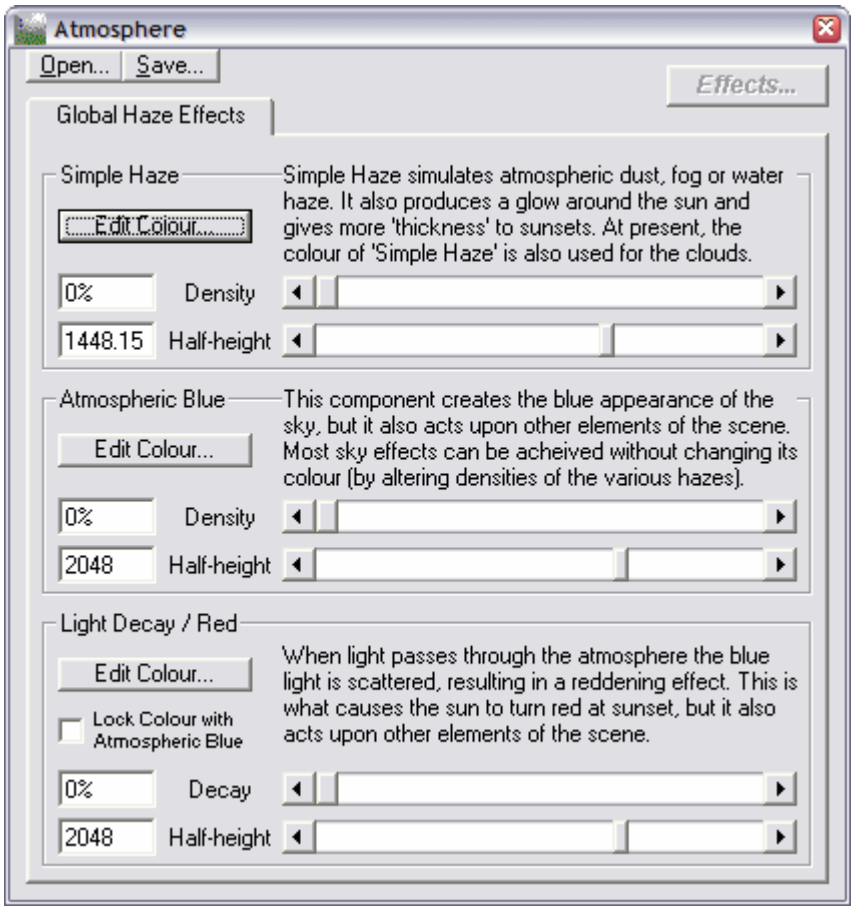
The ‘Camera Orientation’ should already contain the following values, but make sure that the XYZ edit boxes contain (0, -90, 0)

Move the Zoom slider all the way to the right. This helps eliminate perspective, and is why we specified such a large distance above the terrain.

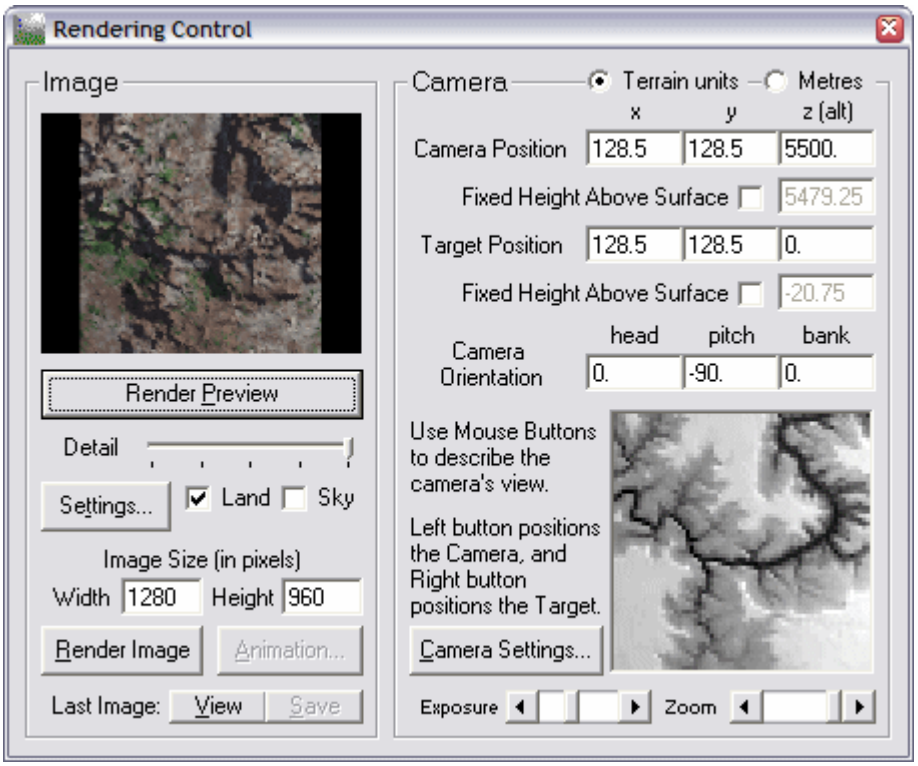
After setting up the camera, we will need to specify how our scene will be lit. For this we need to open up the Lighting Conditions dialog:

The settings you choose within this dialog are largely scene specific, but a heading of around 225 with an altitude of 30 is a good place to start. You can also enable or disable the various shadow casters in this dialog, so if you don’t want the terrain to cast shadows onto itself; then you will need to disable it here. Finally we need to adjust our atmosphere settings so that when our terrain is rendered from such a great height (470,000 feet roughly) it doesn’t appear as if we are looking through a glass of milk ☺ To do this, we need to open up yet another dialog, this time the ‘Atmosphere’ dialog.

To remove the atmospheric effects that will be applied to our image, simply move **both** Density sliders and the Decay slider all the way to the left. We want Haze, Atmospheric Blue and Light Decay/Red values all set at 0% as shown below:



Go back to the Rendering Control dialog. Now set up the image size. In the shareware version of the application we are limited to generating images of a maximum size of 1280x960. This is not a problem because we can scale our terrain to the required size later. In general, the settings used above work best with an image size ratio of 1.33333. At this point, make sure that your detail slider is at maximum (all the way to the right) and save these settings so that you never need to set them again for this terrain. Below we see a screenshot of the Rendering Control dialog in the state it should be in at this point:



As we see above, we rendered a quick preview image. The terrain is viewed from a top-down perspective and is roughly the correct size. All that is left to do now is to render the final image which can be done using the Render Image button. This process can be quite slow if you are using a complex surface map. Fortunately it should only take a few minutes for our examples. You will now be presented with the final terrain image. You should probably save it to file using the Save button in the Image Window.

We still have a bit more work to do before we can use this as a texture in our engine. The resulting image is surrounded by large black borders down either side. To fix this you can simply crop (or trim) the image using your favorite paint package.

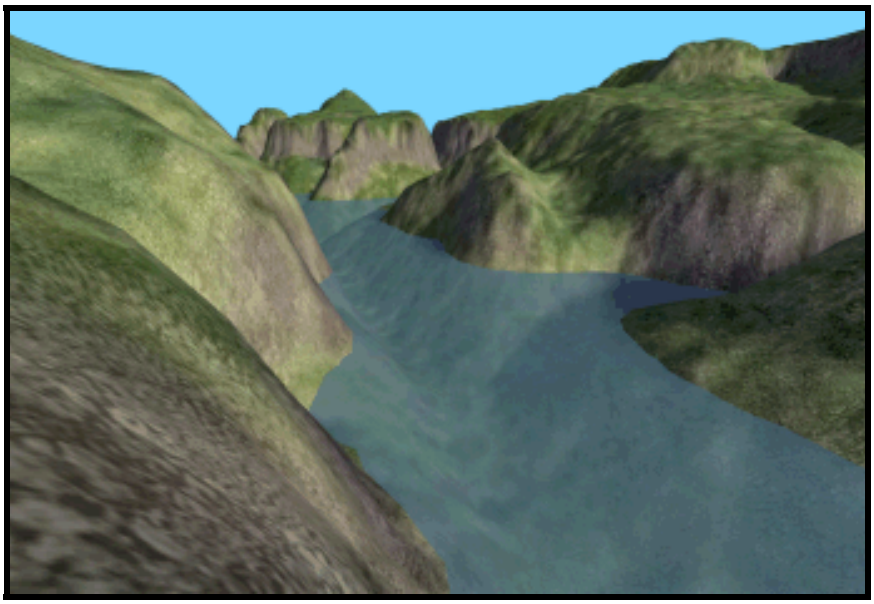
Finally you want to scale the image to its final size, preferably using bilinear or bicubic resampling. We selected a set of nice round numbers at this point and scaled the image to 1024x1024. You can also apply a small amount of blurring to remove any jagged edges if you wish (a Gaussian blur works very well for this purpose). Be careful not to go overboard or you may blur out important surface detail.

Once done, you might need to flip your image (top-to-bottom) because Terragen inverts the heighmap. Simply save the image out again, using your desired format, and it is ready to be applied to the terrain.

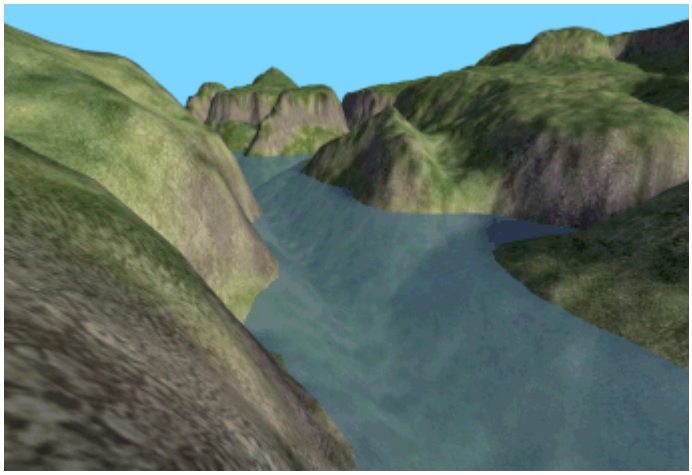
While this all seems like a very complex process, once you set up Terragen and save out the settings, it really just becomes a simple process:

- Load settings
- Import or generate height-map
- Set surface map (if different from that saved)
- Render
- Crop, Size, Save

Workbook Chapter Seven: Alpha Blending and Fog



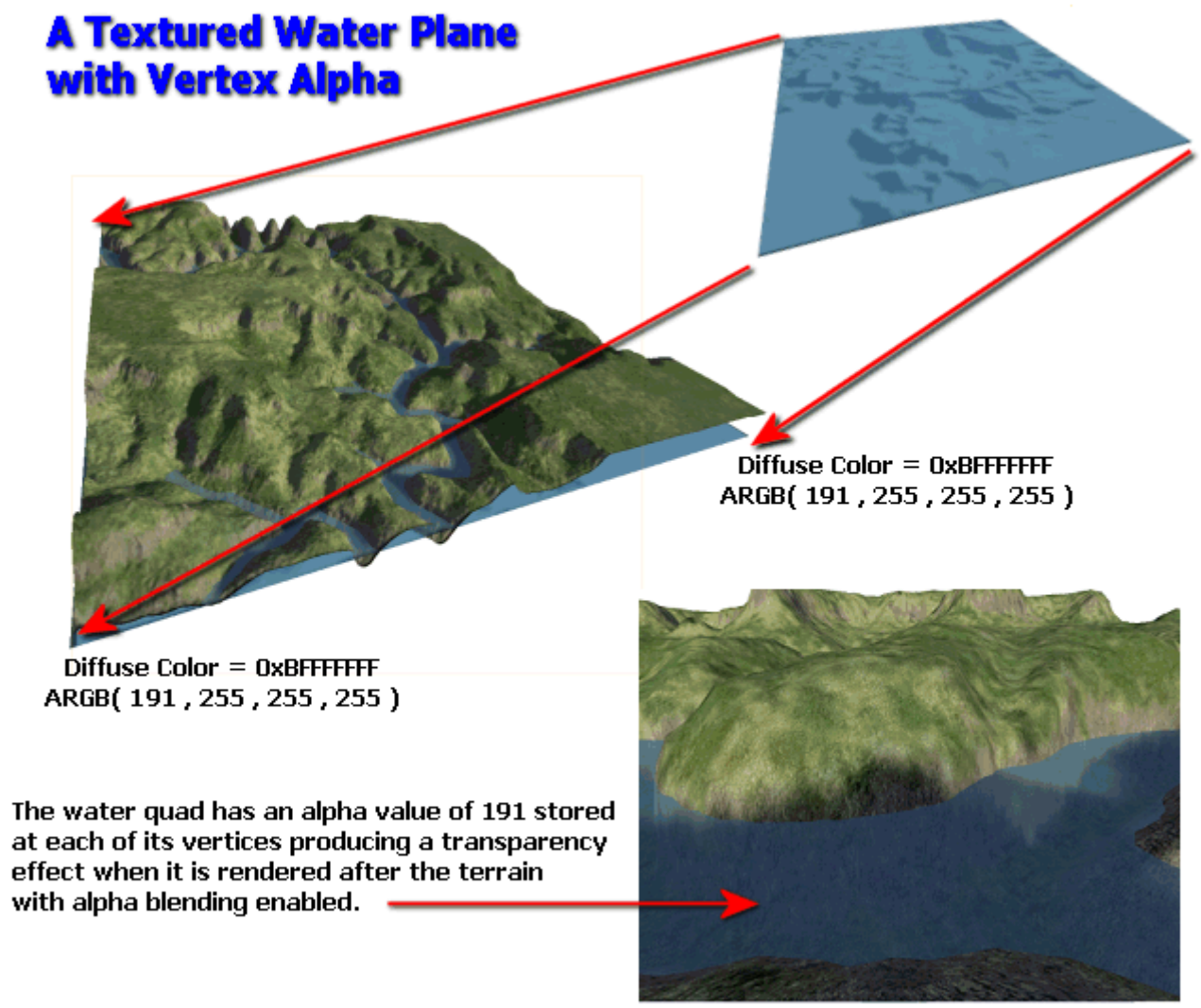
Lab Project 7.1: Vertex Alpha



One of the simplest ways to perform alpha blending is by specifying the alpha component in the diffuse color of the vertex. This color is interpolated across the surface of the polygon at render time to generate a per-pixel color value that is fed into the texture stage cascade. The alpha value is part of the color, so it is treated in exactly the same way as its RGB counterparts. When the polygon is assembled to be rendered, the alpha value is interpolated over the surface to generate a per-pixel alpha that is fed into the texture stage cascade.

Up to this point the alpha and the RGB components of the color have been fused together as a four component color. Interpolating the color automatically interpolated the alpha value along with the RGB values. But once the per-pixel four component color has been interpolated, the per-pixel alpha value is conceptually separated from the RGB components of the color, and the alpha value is sent through the alpha pipeline of the texture blending cascade. RGB components are sent into the color pipeline of the texture blending cascade.

In the above image we can see that the water polygons are transparent so that we can see the terrain through the water. We can increase or decrease how transparent the water is by adjusting the value of the alpha component of the color stored at each vertex. As it turns out, the addition of water to the terrain in this project is really just a matter of adding a single quad to the scene.



In the above image we create an XZ aligned quad that is the same size as the terrain. The water quad has a water texture mapped to it that was also generated in TerraGen™. The quad is placed in the world such that its corners align with the terrain, but is offset by a certain height from the bottom of the terrain.

Since the height of the terrain is 0.0 at its lowest points, placing a water quad at a height of 10.0 means that the water will only be visible where there are troughs in the terrain whose heights are lower than

10.0. Portions of the water quad will be occluded by sections of the terrain that are higher than 10.0. Although this looks odd from the side view in the above picture, when our player is on the terrain they see only the water plane where the terrain dips. This creates a nice collection of rivers and lakes.

We set the color of each vertex in the water quad such that it has an alpha value of 191 (¼ transparent). It is rendered with alpha blending enabled after the terrain has been drawn. The color of each pixel in the quad is sampled from the water texture whilst the alpha for each pixel in the quad is interpolated from the vertex alpha. In this example every vertex has the same alpha value, so we could achieve the same effect by using a Texture Factor color. However, in this project we want to learn how to store and use per-vertex alpha.

There are two new functions added to our CTerrain class from previous chapters. The CTerrain::RenderWater function is called at the end of the CTerrain::Render function to render the water quad after the main terrain has been drawn. The CCamera::RenderScreenEffect function is called at the end of CTerrain::RenderWater and the purpose of this function will become clear shortly.

Rendering the water quad is very easy. We simply build a quad, map the water texture to its four corner vertices and render the terrain with alpha blending enabled. The texture stages are configured to take the alpha values from the vertices of the quad.

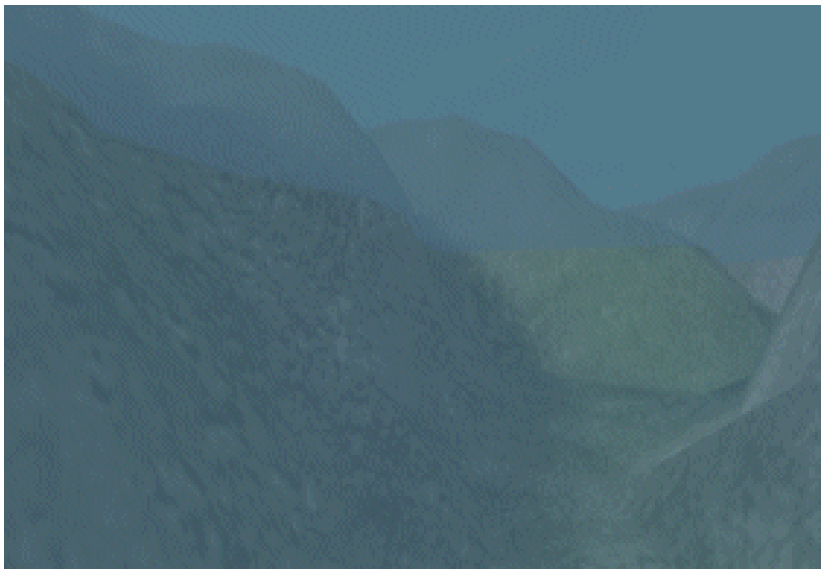
We need to consider what will happen if the player positions the camera underneath the water plane. The image below shows that the illusion of water is lost in this case since the terrain would look the same under the water.

Certainly the terrain should not look the same when we are under the water. At the very least, it should be tinted (blue) since we are viewing it through a volume of (blue) water. There are other effects we can use to enhance our water, but those will have to wait until the next course in this series. To solve our immediate problem we will introduce the use of pre-transformed vertices.

Introduction to Transformed and Lit Vertices

A pre-transformed vertex is a vertex where the x and y components of the vertex are in *screen space* and the z component is in device space. The [0, 1] range for z describes its Z-buffer value. We can use these vertices to bypass the transformation and lighting pipeline and specify polygons in pixel coordinates. They are referred to as transformed and lit vertices (T&L vertices) because we explicitly provide the screen space information and color usually generated by the transformation and lighting pipeline. These vertices can still be used with all of the pixel blending techniques we have discussed, such as alpha blending. However, one thing to note is that the Direct3D fixed function pipeline does not support texture coordinate transformation when using pre-transformed vertices since the transformation pipeline is essentially bypassed.

In our application, we can check to see if the camera is under the water and if so, create a blue screen space quad (using T&L vertices) that covers the entire screen. We then alpha blend it with the contents of the frame buffer after the terrain and water plane have been rendered. This is a cool idea that is extremely easy to implement. The result is shown in the next image.



The device knows whether a vertex needs to be transformed/lit or whether it has been defined in screen space by checking the flexible vertex format flags used to create the vertex.

> **Note**: Transformed and lit vertices are useful if you are converting a software engine to DirectX. In that case, the transformation pipeline has already been programmed, so DirectX would be used only for rendering the polygon.

When creating a screen space vertex, we need the x and y positions describing the screen space location of the vertex in pixel coordinates and a z value between 0.0 and 1.0 that describes the Z-Buffer space vertex distance. The latter is necessary because the Z-Buffer test is still performed for

T&L vertices (unless specifically disabled). We also need a 1/W value for W-Buffer and fog calculations. 1/W is called the reciprocal of homogeneous W (RHW). As discussed in Chapters 1 and 2, the W component is usually just the same as the view space Z coordinate of the vertex, so RHW will also be a value between 0.0 and 1.0. A vertex with a higher RHW value will be regarded as being closer to the camera. Each vertex will usually have a diffuse color and may also include a specular component. The rasterizer will add these two colors together and interpolate them for each pixel rendered.

The CTLitVertex Class

We will create an additional vertex class to store transformed and lit vertices used to render the screen space quad (which we will call the ScreenEffect). We will not need texture coordinates or specular color as we will simply be alpha blending a blue quad on the screen. We will need four floats to hold the X, Y, Z, and RHW (1/w) components and a diffuse (blue) color. The alpha value specified with this color will describe how transparent the screen space quad should be.

```
class CTLitVertex
{
public:
    //-----------------------------------------------------------------------
    // Constructors & Destructors for This Class.
    //-----------------------------------------------------------------------
    CTLitVertex ( float fX, float fY, float fZ, float fW, ULONG ulDiffuse = 0xFF000000 )
                          { x = fX; y = fY; z = fZ; w = fW; Diffuse = ulDiffuse; }

    CTLitVertex()  { x = 0.0f; y = 0.0f; z = 0.0f; w = 0.0f; Diffuse = 0xFF000000; }

    //-----------------------------------------------------------------------
    // Public Variables for This Class
    //-----------------------------------------------------------------------
    float       x;                  // Vertex Position X Component
    float       y;                  // Vertex Position Y Component
    float       z;                  // Vertex Position Z Component
    float       w;                  // Vertex Position W Component
    DWORD       Diffuse;    // Diffuse vertex color component
};
```

We will use the IDirect3DDevice9::SetFVF function to inform the device of the structure of our vertices before using them to render the screen effect. We do this with the flexible vertex formats flags shown below (defined in CObject.h).

```
#define TLITVERTEX_FVF  D3DFVF_XYZRHW | D3DFVF_DIFFUSE
```

When the device encounters the D3DFVF_XYZRHW flag it will not to transform or light the vertices. It assumes that the x and y components of the vertex are already in screen space. Thus, defining a quad that completely covers the screen (using a triangle fan) can be done with the simple code shown below:

```
CTLitVertex TopLeftVertex ( 0.0 , 0.0 , 0.0 , 1.0 , 0x800000FF);
CTLitVertex TopLeftVertex ( ViewPortWidth , 0.0 , 0.0 , 1.0 , 0x800000FF);
CTLitVertex TopLeftVertex ( ViewPortWidth , ViewPortHeight , 0.0 , 1.0 , 0x800000FF);
CTLitVertex TopLeftVertex ( 0.0 , ViewPortHeight , 0.0 , 1.0 , 0x800000FF);
```

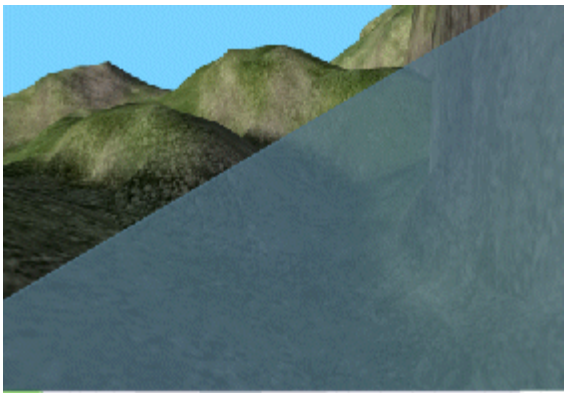
> **Note**: Remember that in screen space increasing Y moves downwards towards the bottom of the screen.

Since we want the screen quad to be rendered on the top of everything else in the frame buffer, we set the Z value of the vertex to 0.0. This will put it right at the front of the Z-Buffer on the near clip plane. We set the RHW value to 1.0 so that if we are using a W-Buffer or W-based fog (covered later), the distance is also calculated correctly. Remember RHW is 1/w where w is usually the view space Z coordinate. An RHW value of 1.0 means the Z view space Z distance was 0. An RHW value of 0.0 means the vertex is at the back of the frustum on the far clip plane. When using a W-Buffer instead of a Z-Buffer it is this RHW value that is used for depth testing instead of the Z value. This gives a more artifact free rendering of the scene over a far distance. Although W-Buffers have been largely forgotten now that graphics cards support 24 and 32-bit Z buffers, the RHW value is still used to calculate the distance to the vertex for fog effects.

The vertices defined in the above code render a blue quad over the entire viewport. Each vertex also contains a ½ intensity alpha value for later blending. This is exactly the approach we will use in our project.

Handling Partial Submersion

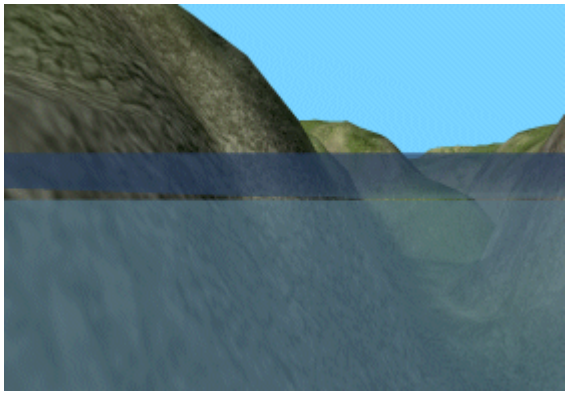
It is possible for the camera to be positioned either just above the water line or just below it. For example, the water line may be positioned half-way up the screen. As the player starts to sink into the water there comes a point at which half of their view is above the water line and half is below. In that case we would want the bottom half of the frame buffer to have the blue screen space quad blended over it, but not the top half. One might assume that we could simply take the height of the water in world space and convert this into screen space and use it as the height from the bottom of the screen for the quad. While this would work under certain circumstances, our camera can bank (roll) left and right and this approach would not produce correct results. So we cannot rely on the fact that the screen effect can even be rendered as a simple quad. The following image shows the camera positioned such that the viewer can see both above and below the water line at the same time. Note that the camera is also banked.



We will need to calculate the slope of the water line with relation to our camera and build a screen space polygon with the same slope so that it connects properly to the water plane itself.

So let us now revise what we need to do to accomplish our water effect:

1. Render terrain
2. Render water quad using alpha blending
3. Calculate the slope vector of the water on the near plane
4. Create a screen space polygon with the correct height and slope so that it aligns with the slope of the water quad on the near plane.



Item number 4 in the above list is slightly problematic. On some drivers and on some hardware we noticed that calculating the slope of the water line for our screen effect did not perfectly line up with the water plane that the DirectX transformation pipeline had rendered and clipped. This is no doubt due to epsilon errors or some other clipping optimization that makes the clipping slightly less accurate than it could be. The following image shows the problem that occurs when we use the DirectX pipeline to clip our water plane to the near plane.

You can see gaps caused by the slight misalignment between the quad and the water polygon. To address this, we will do the clipping of the water plane to the near plane of the frustum ourselves to ensure accuracy. We also want to be sure that when we calculate the screen space height and slope of our screen effect quad, that it lines up exactly with the slope of the water polygon being clipped to the near plane.

The revised to-do list now looks like this:

1. Render Terrain
2. Clip water polygon to near plane of frustum

3. Render water polygon
4. Calculate the height and slope of the water polygon on the screen
5. Create a screen space polygon that has a top edge that matches this height and slope

Classifying a point against a plane

As mentioned above, we will have to clip the water plane to the near plane of the frustum ourselves because the pipeline may not have done a sufficient job. We need this clipping to be exact so that the water plane and the screen effect line up exactly on the screen.

Clipping a polygon to a plane is a relatively easy procedure. The first thing we need to be able to do is classify a point against a plane. This informs us of the distance to the plane from the point. More importantly in this case, the distance is a signed result which we can interpret to indicate whether the point is in front of the plane or behind the plane for any point where the distance is not zero. If the distance from the point to the plane is zero, then the point is said to lie on the plane.

> **Note:** In Chapter 1 we learned that a plane consists of a 3D unit vector describing the plane normal (A,B,C) and a distance value D describing the distance to the plane from the origin. Thus, we could store a plane using a 4D vector (where x,y,z,w map to A,B,C,D). However, the D3DX library includes a D3DXPlane structure defined specifically for storing planes:

```
typedef struct D3DXPLANE
{
   FLOAT a;
   FLOAT b;
   FLOAT c;
   FLOAT d;
} D3DXPLANE;
```

We will need to make use of the Plane Equation to accomplish our objective. The equation can actually take one of two forms:

Plane Equation 1: $Ax + By + Cz + D = 0$
Plane Equation 2: $Ax + By + Cz - D = 0$

Which one you use depends on whether the distance (D) is considered positive or negative when the origin is behind the plane. Let us have a look at this in a bit more detail.

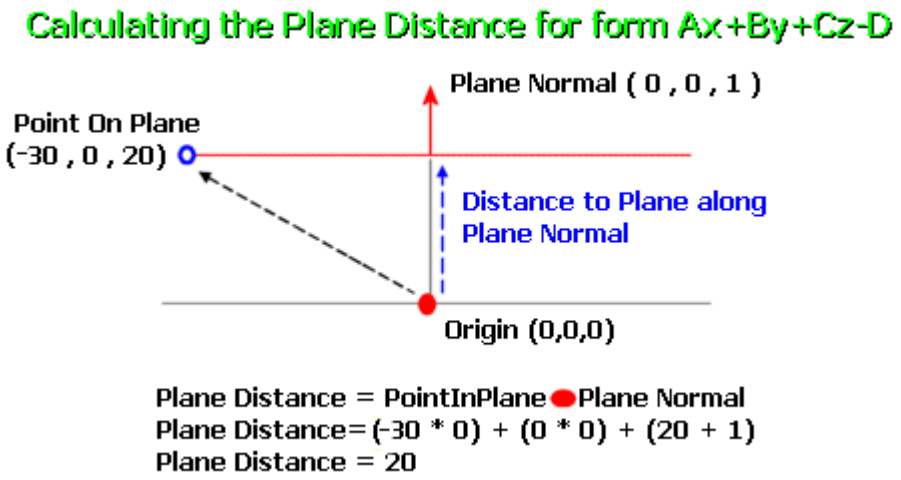
To calculate the plane distance we perform a dot product between the plane normal and a point known to be on the plane. This returns a value that is negative if the origin is in front of the plane and positive when behind. When using the distance in this manner, we need to use Plane Equation 2 to classify a point in space against the plane. Often this is the more intuitive of the two. If the origin is behind the

plane, then the distance will be positive. Remember that this is the distance you would have to travel along the plane's normal vector from the origin to be at a point that was on the plane.

However, many functions (including the D3DX helper functions) use Plane Equation 1. When using this form of the plane equation, the distance will be negative when the point is behind the plane and positive when it is in front. In this case the distance no longer describes the length of travel down the plane normal, but instead describes the length of travel down the reverse of the plane normal. This may be a little less intuitive but ultimately it is a matter of preference.

Let us quickly look at using the two different plane representations. Remember that the only difference between the two is that equation 1 expects the plane distance to be positive when the origin is in front of the plane and equation 2 expect the plane distance to be negative when the origin is in front of the plane.

If we use equation 2, we calculate the distance as follows:

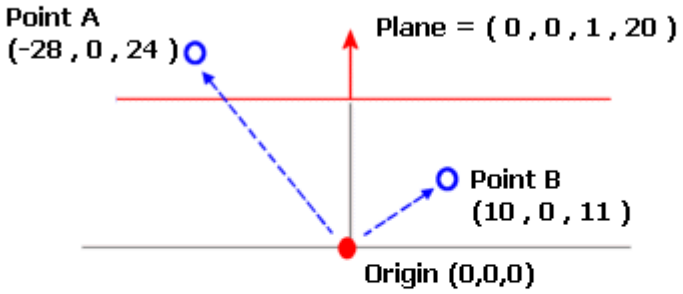


In this example we have a plane normal that is pointing down the positive Z axis and a point known to be on the plane (a polygon vertex for example). We perform the dot product between this point and the plane normal and get back the distance to the plane from the origin along the plane normal. We can see that the plane is at a distance of 20 units from the origin. Using this method, the distance is positive when the origin is behind the plane (on the opposite side of the plane to which the plane normal is facing) and negative if the point is in front of the plane.

(0,0,0) + (0,0,1) * Distance = Point on Plane nearest to origin
(0,0,0) + (0,0,1) * 20 = (0 , 0 , 20)

When we have our planes stored in this form, we can classify any point in space against the plane to find the distance of the point to the plane. We use the plane equation Ax+By+Cz-D as shown in the following diagram:

Classification of points using the form Ax+By+Cz-D



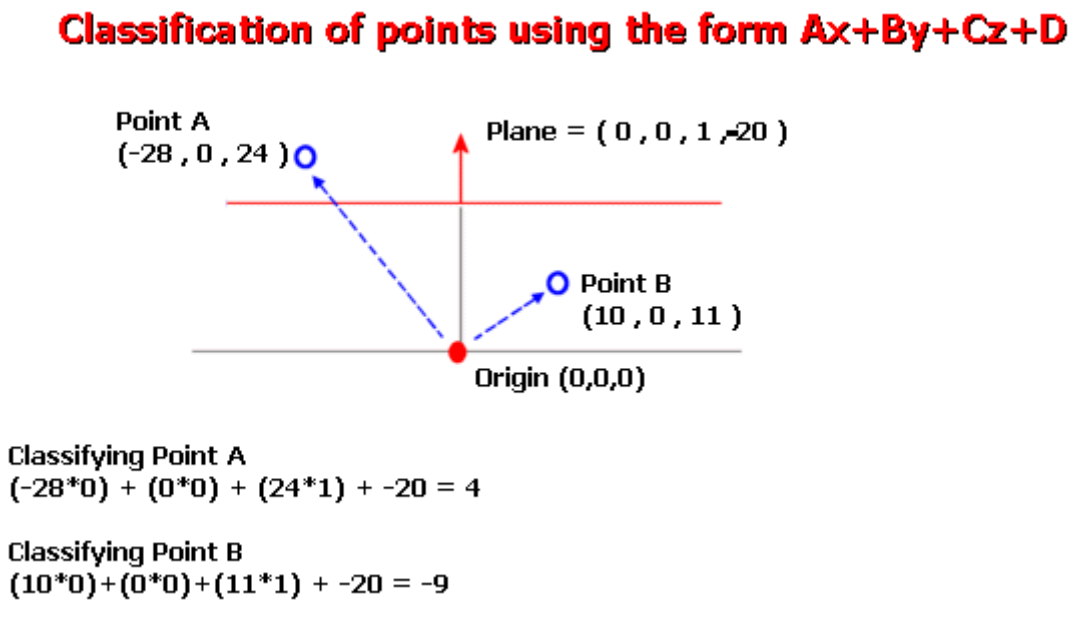
Plane Equation = Ax + By + Cz -D
where xyz = point we wish to classify
and ABCD = the plane components

Classifying Point A
(-28*0) + (0x0) + (24*1) - 20 = 4 (Positive Distance)

Classifying Point B
(10*0) + (0*0) + (11*1) - 20 = - 9 (Negative Distance)

In this equation we subtract the plane distance from the dot product of the point and the plane normal. If the point is in front of the plane, the distance from the point to the plane will be a positive value. If the point is sitting on the plane then the result will be zero. Finally if the point is behind the plane the result will be negative. You can see that Point B is at a distance of 9 units from the plane. Because it is behind the plane, its distance is negative.

The other form of the plane equation is Ax+By+Cz+D. Planes can have the sign of their distance component flipped such that the plane distance is positive if the origin is in front of the plane and negative if the plane distance is behind the plane. The D3DX function D3DXPlaneDotCoord function classifies points against planes using this approach. The results are interpreted the same way as the first approach (negative if the point is behind or positive if the point is in front) but the plane distance has the opposite sign. So we will have the plane stored with a distance of –20 units in this case because origin is behind the plane instead of in front of it.



Because we will be using the Ax+By+Cz+D form of the plane equation, we must make sure that our planes have positive distance values if they are facing the origin.

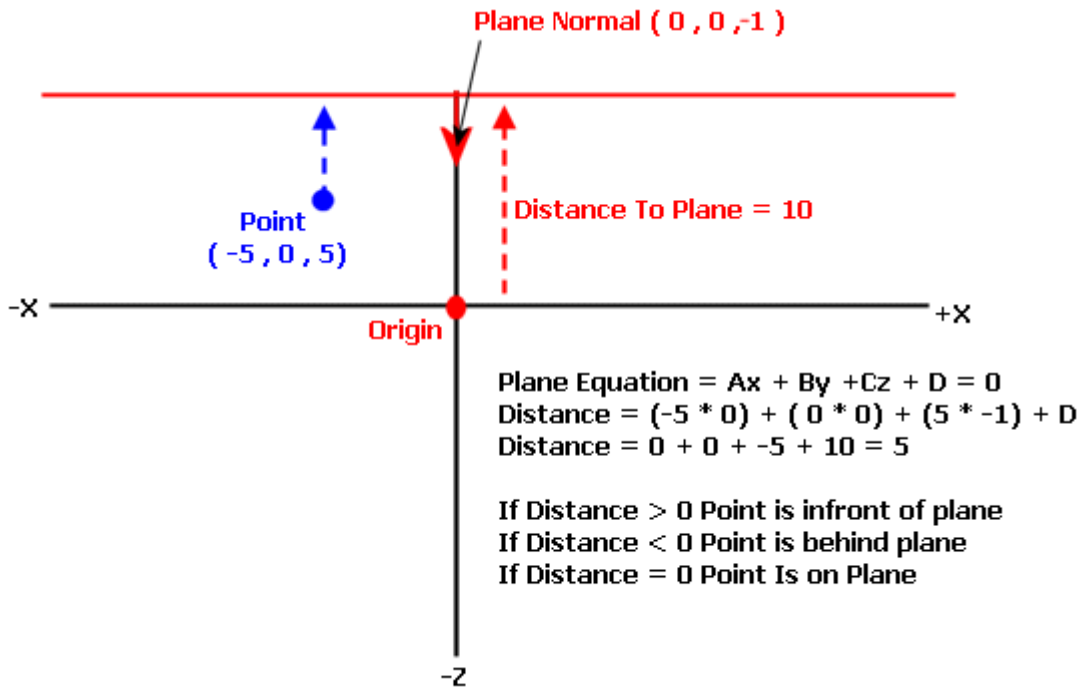
The D3DXPLaneDotCoord function is shown below:

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

The function takes the address of a D3DXPLANE structure and the address of a 3D vector representing the point to be classified. The 4th component of the point will be treated as a 1.0 (making it a homogeneous coordinate) sp that a 4D dot product can be performed:

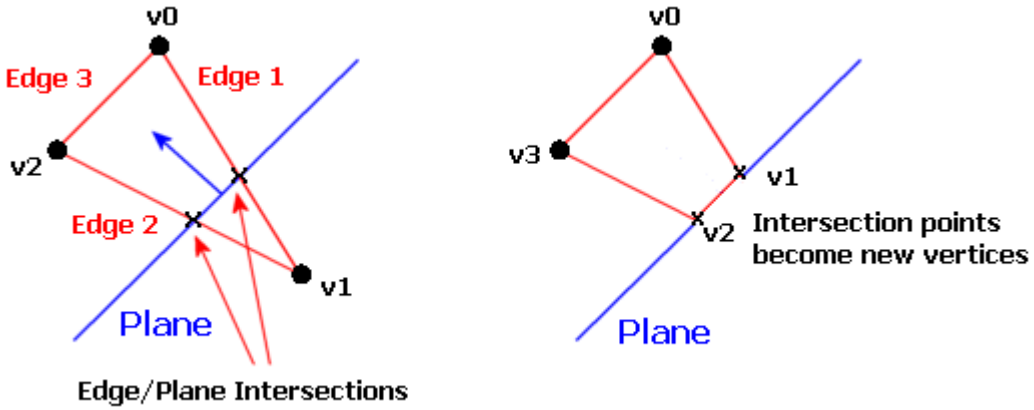
```
FLOAT D3DXPlaneDotCoord ( D3DXPLANE * pP , D3DXVECTOR3 *pV)
{
    return ( pP->x * pV->x ) +  ( pP->y * pV->y ) + ( pP->z * pV->z ) + (pP->d *  1.0)
}
```

We can interpret the result of this function as shown below. Remember that we should use a positive plane distance if the plane normal is facing the origin.



Point/Plane classification is critical because it will determine how we clip polygons to a plane. In this particular demo we want to clip the water polygon to the near plane of our viewing frustum in world space. But the technique can be used in other situations as well where clipping is needed.

Let us assume that we wish to clip a triangle against a plane. We can loop through each vertex in the polygon and test to see if it is in front, behind, or on the plane. Any vertex that is behind the plane will be clipped. New vertices will occur at the points of intersection as shown below:

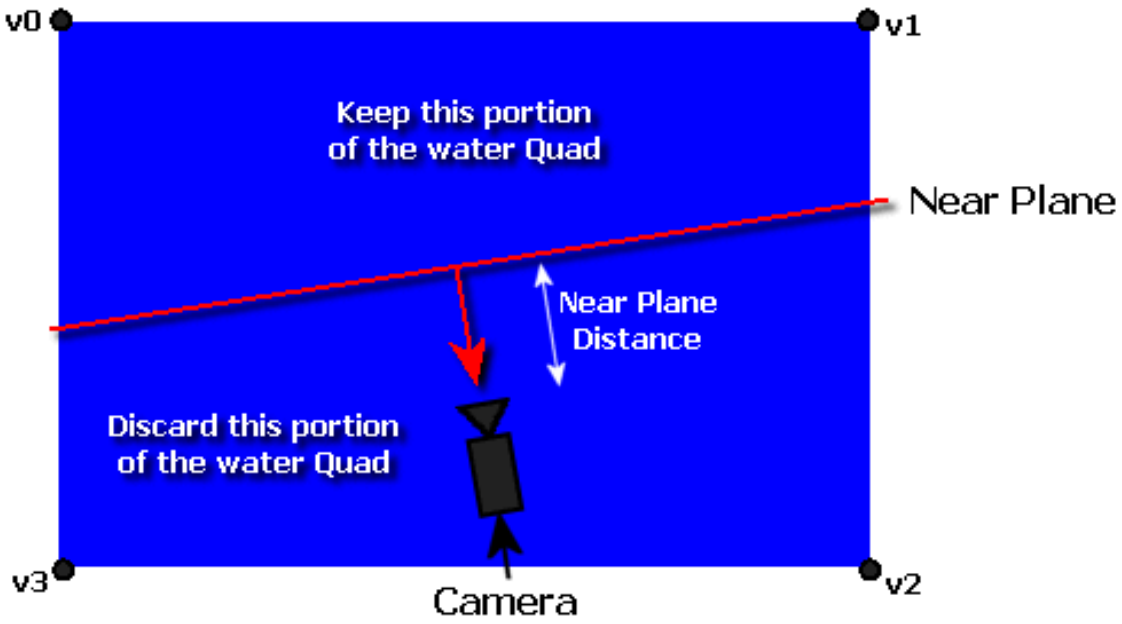


Thus the procedure for each polygon to be clipped would be:

1. Test each edge of the polygon by classifying each point in the edge against the plane
2. If both points of the edge are in front of the plane then keep the two vertices and this entire edge because it is not clipped by the plane at all (Edge 3 in the above example).
3. If both vertices in the edge are behind the plane then the edge should be completely clipped. Neither of these vertices will exist in the new clipped polygon.
4. If one of the points in an edge is in front of the plane and another is behind the plane then we have an edge that is spanning the plane. When this is the case we must calculate the point on the plane where the edge intersects the plane. This point will become a new vertex which will replace the vertex that is behind the plane. In the above example we see that Edge 1 is spanning the plane. The intersection point is calculated and added to the new polygon and the original v1 is discarded. The same happens when we check Edge 2, which is also spanning. v1 has already been rejected but we still need to calculate the intersection point with the plane and add it to the clipped polygon.
5. We do this for each edge in the original polygon, building a new polygon as shown in the right hand image.

The clipped polygon does not always have the same vertex count as the pre-clipped polygon, and new edges can be introduced on the plane (v1-v2 in the right diagram above). While this may sound complicated, it is actually quite simple to do. We will simply treat our edges as 3D lines and use a line/plane intersection test to find points of intersection. We will discuss this test shortly.

Now that we understand at a high level how to clip a polygon to a plane, we can start to see how we can clip our water polygon to the near plane. The near plane will have a normal that is facing the camera in world space. Because of this, we will need to clip the water quad to the near plane such that only the area of the quad that exists behind the plane survives.



In order to study the clipping procedure in more detail, let us start analyzing some of the source code to Lab Project 7.1.

CTerrain::RenderWater

The RenderWater function builds, clips, and renders the water quad. We define a constant called WaterLevel which contains the height of the water polygon on the terrain.

```
const float  WaterLevel  = 54.0f;
```

This height value is defined in height map space just as the height of each pixel in the height map is. As such the first thing the function must do is use the terrain scale to convert the water level into the world space height of the water polygon.

```
void CTerrain::RenderWater( CCamera * pCamera )
{
    CLitVertex  Points[5];
    int         PointCount = 0;

    // Retrieve floating point world space water height
    float WaterHeight = WaterLevel * m_vecScale.y;
```

If we are well above the water then we can simply render the water quad as is. We only need to clip it if we are going to need to do an underwater screen effect. So in our demo, when the height of the camera is more than 10 world units above the water plane, there is no way that camera can see underwater and therefore no need to run the clipping operation. When this is the case then the quad is simply built and rendered. If the camera is even partially underwater however we need to clip the water polygon to the near plane. The Points array allocated at the top of the function will be used to hold the vertices of the clipped water polygon.

```
    // If we are close enough to the water, we need to clip
    if ( pCamera && (pCamera->GetPosition().y - 10.0f) < WaterHeight )
    {
        // Build a combined projection / view matrix
        D3DXMATRIX mtxCombined = pCamera->GetViewMatrix() * pCamera->GetProjMatrix();

        // Extract the near clipping plane.
        D3DXPLANE NearPlane;
        NearPlane.a = -(mtxCombined._13);
        NearPlane.b = -(mtxCombined._23);
        NearPlane.c = -(mtxCombined._33);
        NearPlane.d = -(mtxCombined._43);
        D3DXPlaneNormalize( &NearPlane, &NearPlane );
```

We extract the near plane in accordance with our discussion in Chapter Four. When the information is extracted from the matrix it is not normalized. However, a plane that has not been normalized can still be used with the plane equation to classify whether a point is in front or behind because the sign of the value returned will still be the same (although the distance value will not be). We have nevertheless decided to normalize the plane normal because we may need it later to correctly calculate the distance.

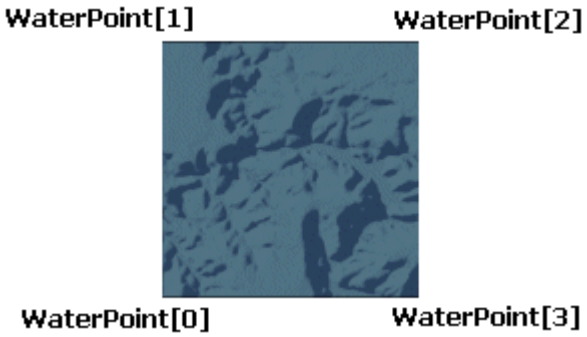
The near plane is defined when we setup the projection matrix. It will always have a plane normal that points in the opposite direction of our look vector. Therefore we can think of our camera look vector as

always pointing at the near plane (which is at some distance in front of the camera). In turn we can think of the near plane normal as pointing back at the camera in the opposite direction as shown in the previous diagram. So we will clip away any part of the water plane that is in front of the near plane and keep any vertices that are behind it.

Our next task is to build the initial water quad. This quad is built from four pre-lit, untransformed vertices -- the same as the type used in the terrain itself. For simplicity we will use the DrawPrimitiveUP function since we only have to quickly build and render a single quad. Nevertheless, a vertex buffer approach would be preferred and at the end of the lesson, you should try this as an exercise.

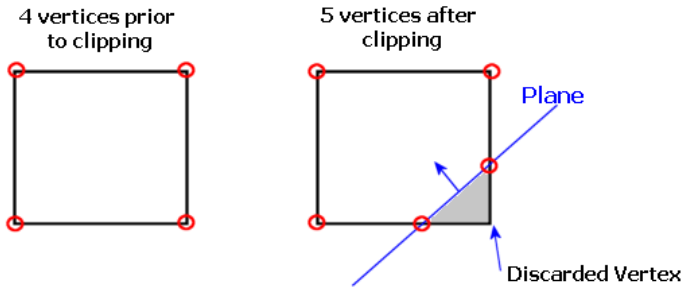
Our vertices will define the four corner points of the water polygon and they are placed at the four corner points of the terrain. Each vertex has a Y value equal to the desired height of the water level. A white diffuse color with an alpha value of 191(0xBF) is also included for transparency. The RGB color will not be used since the quad will have a water texture to provide color. Thus we will need to set the texture coordinate at each vertex so that each corner of the quad is mapped to the relevant corner of the texture.

```
        // Build initial 4 corner vectors
        CLitVertex vecWaterPoints[4];

        vecWaterPoints[0] = CLitVertex( 0.0f, WaterHeight, 0.0f, 0xBFFFFFFF, 0.0f, 0.0f );

        vecWaterPoints[1] = CLitVertex(0.0f, WaterHeight, m_nHeightMapHeight*m_vecScale.z,
                                       0xBFFFFFFF, 0.0f, 1.0f);

        vecWaterPoints[2] = CLitVertex( m_nHeightMapWidth * m_vecScale.x, WaterHeight,
                                        m_nHeightMapHeight * m_vecScale.z, 0xBFFFFFFF,
                                        1.0f, 1.0f );

        vecWaterPoints[3] = CLitVertex( m_nHeightMapWidth * m_vecScale.x, WaterHeight,
                                        0.0f, 0xBFFFFFFF, 1.0f, 0.0f );
```



Now we have our near plane in world space as well as our big water quad, so we can now run our clipping operation. The resulting polygon will be stored in the CLitVertex array ‘Points’ allocated at the beginning of the function. Notice that we allocated enough space for five vertices rather than four

as you might have expected. When a quad is clipped, the resulting polygon can have an extra vertex added when two edges connecting to the same point intersect the plane (see diagram).

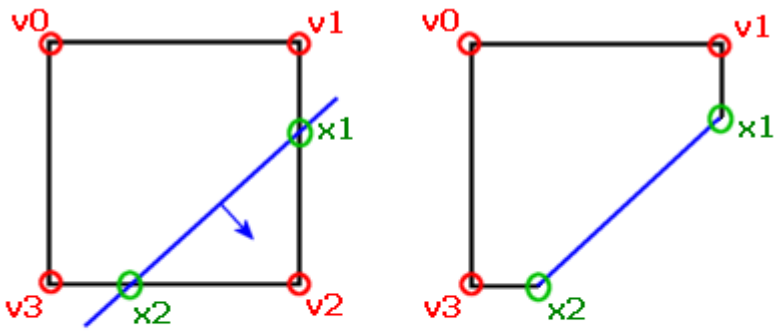


The image on the left shows the quad before clipping and the image on the right shows the resulting polygon. Clipping a quad to a single plane can only introduce one new vertex (at most) in the resulting polygon.

So we will loop through each vertex in the quad and classify it against the plane. If it is on the plane then it must exist in the resulting polygon, so it is added straight away. We also classify the next vertex because these two vertices will form an edge. We then classify the second vertex in the edge against the plane so that we know the location of each vertex in the edge with respect to the plane.

Let us walk through a quad clipping example in theory first before we write at the actual code.

We loop through each vertex in the source polygon starting with vertex 0. Since it is behind the plane we will keep it and it is added to the clipped polygon vertex list as the first vertex. Before finishing up the first iteration of the loop, we need to test that v1 is not on the opposing side of the plane to v0. In that case the edge spans the plane and additional work has to be done. In this example this is not the case, so we continue with the next loop iteration. v1 is tested and is also behind the plane so it is copied into the clipped polygon's list. There are now two vertices in the clipped polygon vertex list. Before finishing this iteration of the loop we check if the next vertex (v2) is on the opposite side of the plane. Since this is indeed the case, the edge formed by v1 and v2 needs to be clipped. Our next job is to calculate the point on the plane where the edge intersects the plane. Once found, this vertex (x1) is added to the clipped polygon list and we have finished with the second iteration of the loop. We move on to the third iteration where we test v2. This vertex is in front of the



Source Polygon
v0
v1
v2
v3

Clipped Polygon
v0
v1
x1
x2
v3

plane so it should not be added to the clipped polygon and we can skip it. Again, before leaving the current iteration of the loop we must check that the next vertex in the loop is not on the opposite side of the plane. Quite clearly we see that v2 is in front but v3 is behind and as such, a new vertex has to be added at the intersection point. As before, we calculate the point at which the edge v2-v3 intersects the plane (x2) and add it to the clipped polygon list. The clipped polygon now has four vertices in its list (v0, v1, x1, x2). Finally we enter the fourth and final iteration of the loop where we test v3 against the plane. v3 is behind the plane so it is added to the clipped polygon vertex list. We also check that the next vertex in the list (which has looped back round to v0) is not on the opposite side of the plane. However, v3 and v0 are both on the back side of the plane so our work is done.

We can now look at the main loop in this function. It classifies each vertex in the source quad against the clip plane and copies them over into the clipped polygon array when they are behind the plane.

```
        // Clip this quad against the plane, discard anything in front
        for ( int v1 = 0; v1 < 4; v1++ )
        {
            int v2 = (v1 + 1) % 4;

            // Classify each point in the edge
            int Location1 = 0, Location2 = 0;

            float result = D3DXPlaneDotCoord(&NearPlane,(D3DXVECTOR3*)&vecWaterPoints[v1]);
            if ( result < -1e-5f ) Location1 = -1; // Behind
            if ( result >  1e-5f ) Location1 =  1; // In Front

            // Keep it if it's on plane
            if ( Location1 == 0 )
            {
                Points[ PointCount++ ] = vecWaterPoints[v1];
                continue; // Skip to next vertex
            }

            result = D3DXPlaneDotCoord( &NearPlane, (D3DXVECTOR3*)&vecWaterPoints[v2] );
            if ( result < -1e-5f ) Location2 = -1; // Behind
            if ( result >  1e-5f ) Location2 =  1; // In Front
```

The line that assigns a value to v2 basically says “let v2 equal v1+1 unless v1 is the last vertex in the quad, in which case v2 will be set to zero”. This is because the edges of the quad are formed by the vertices as shown below.

Edge1 v0 → v1
Edge2 v1 → v2
Edge3 v2 → v3
Edge4 V3 → v0

So when processing edge 4, we need to loop back around to the first vertex. Note that although we are actually checking two vertices here, it is only the first vertex that will either be rejected or added to the resulting polygon. The second vertex in the edge is used only to determine whether the edge spans the plane. The next line checks if the current vertex is behind the plane and adds it to the new vertex array for the resulting clipped polygon if it is.

```
            // If it's not in front, keep it.
            if ( Location1 != 1 ) Points[ PointCount++ ] = vecWaterPoints[v1];
```

If the second vertex in the edge is either on then plane or on the same side of the plane as the first vertex then it means the edge formed by these two vertices is not spanning. We can then continue on to the next iteration of the loop where the second vertex from this iteration will become the first vertex of the edge in the next iteration.

```
    // If the next vertex is not causing us to span the plane then continue
    if ( Location2 == 0 || Location2 == Location1 ) continue;
```

If the second vertex is on the opposite side of the plane then the edge is spanning and we need to calculate the intersection point. This point will be come a new vertex in the polygon -- in place of the vertex that was in front of the plane.

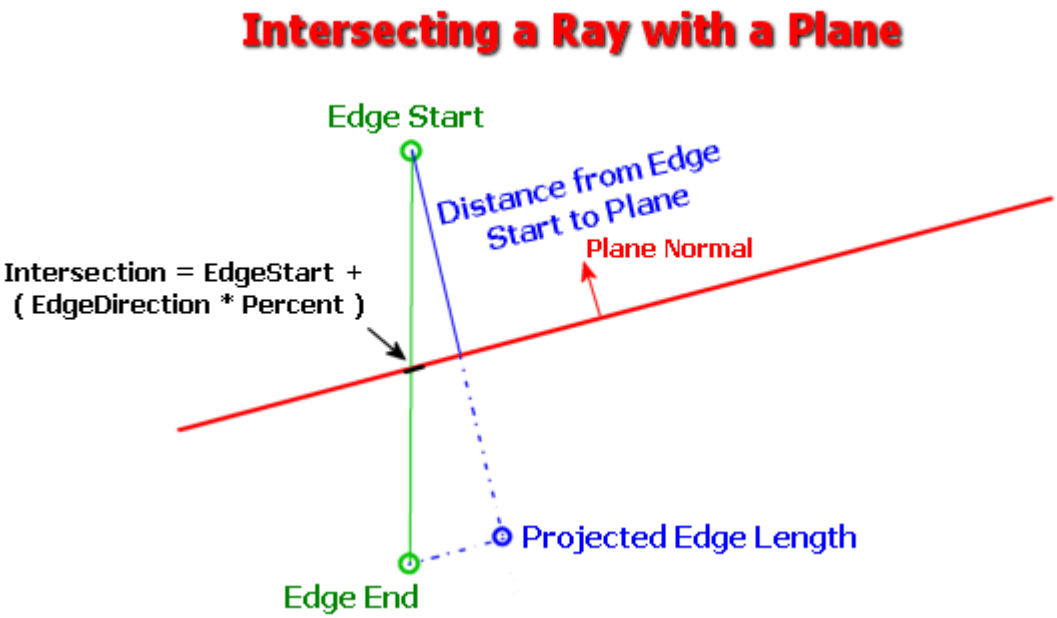
The D3DX library includes a function called D3DXPlaneIntersectLine which take two edge points (the two vertices of the edge we are processing) and a plane and returns the intersection point as a 3D vector. We can simply call this function using our test vertices and retrieve the intersection point containing the new x,y,z coordinates of the vertex we need to insert which lies on the plane (x1 or x2 from the previous diagram).

```
D3DXVECTOR3 *D3DXPlaneIntersectLine
(
   D3DXVECTOR3 *pOut,
   CONST D3DXPLANE *pP,
   CONST D3DXVECTOR3 *pV1,
   CONST D3DXVECTOR3 *pV2
);
```

While D3DX obviously makes life easier for us, this is a test worth examining since it is rather common and you may need to implement your own line/plane intersection routines in the future should you find yourself programming in an environment where D3DX is not available. So let us take a quick detour just to explain how this function works and then return to covering the rest of the CTerrain::RenderWater function.

Line/Plane Intersection

Our line will be defined by a start and end point in 3D space. We will treat each edge of the polygon as a line and perform a line/plane intersection. The process is quite easy to understand provided we have a thorough understanding of the dot product.



In the above image you can see that *EdgeStart* and *EdgeEnd* points could be the two vertices of the edge that we are testing. If we subtract *EdgeStart* from the *EdgeEnd* we get a new vector that describes the direction of the line from *EdgeStart* to *EdgeEnd.* The length of this vector describes the distance from *EdgeStart* to *EdgeEnd*. We will call this vector EdgeDirection where:

EdgeDirection = EdgeEnd – EdgeStart
EdgeEnd = EdgeStart + EdgeDirection

Now we must determine where the plane intersects this line. If the line were perfectly aligned with the plane normal we could easily determine this by performing a classification of the EdgeStart point against the plane. This would return the distance to the plane shown as the solid blue line in the above diagram. While this is not the case, we can still use this value. With the distance to the plane along the plane normal, we can calculate how long our EdgeDirection line would be if it was rotated to be aligned with the plane normal. We can determine this by performing the dot product between the EdgeDirection line (the green line) and the plane normal. This will scale the EdgeDirection line by the cosine of the angle between the plane normal and the EdgeDirection vector and give us the projected length of the EdgeDirection vector along the normal. In other words, this would give us the distance from the EdgeStart point to the EdgeEnd point if the line had been rotated to be aligned with the plane normal.

Because we know the distance from the EdgeStart point to the plane and we know the distance from the EdgeStart point to the Projected EdgeEnd point along the plane normal, dividing the Projected EdgeEnd point by the Distance to the plane from the EdgeStart point will return a value between 0.0 and 1.0. This value describes how far along the line the intersection has happened (where 0.0 is the EdgeStart point and 1.0 is EdgeEnd). If the plane intersected the line exactly halfway between EdgeStart and EdgeEnd this value would be 0.5. Because this value describes the intersection as a

percentage, we have called the variable that receives the result 'Percent' in the following code. Once we know how far along the projected line (the line aligned with the normal) the intersection occurs we know that the intersection along the real line occurs at the same place. Therefore we can use the parametric form of a line to determine the final point of intersection:

Intersection = EdgeStart + (EdgeDirection * Percent)

EdgeDirection is the vector describing the direction and magnitude of the line and percent describes how far along this line (between 0.0 and 1.0) the intersection happens. The following function shows how one might write a LinePlaneIntersection function that works the same way as the D3DXPlaneIntersectLine function

```
D3DXVECTOR3 * PlaneIntersectLine (D3DXVECTOR3 *Intersection , D3DXPLANE * Plane,
                                  D3DXVECTOR3 *EdgeStart, D3DXVECTOR3 *EdgeEnd)
{
    D3DXVECTOR3 Normal;
    Normal.x = Plane->a ;  Normal.y = Plane->b ;  Normal.z = Plane->c;
    D3DXVECTOR3 EdgeDirection          = *LineEnd – *LineStart;
    float       ProjectedEdgeLength    = D3DXVec3Dot ( &Direction , &Normal );
    float       DistanceToPlane        = D3DXVex3Dot ( EdgeStart , &Normal ) + Plane->d;
    float       Percent                = DistanceToPlane / EdgeLength;

     *Intersection = *EdgeStart + ( EdgeDirection * Percent );
}
```

Let us now return to our RenderWater function. We left off when we found an edge intersecting the plane. This means we need to add the intersection point to the vertex list of the clipped polygon so we call D3DXPlaneIntersectLine to calculate this point:

```
// Calculate the intersection point
D3DXVECTOR3 vecIntersection;
D3DXPlaneIntersectLine( &vecIntersection, &NearPlane, (D3DXVECTOR3*)&vecWaterPoints[v1],
                        (D3DXVECTOR3*)&vecWaterPoints[v2] );
```

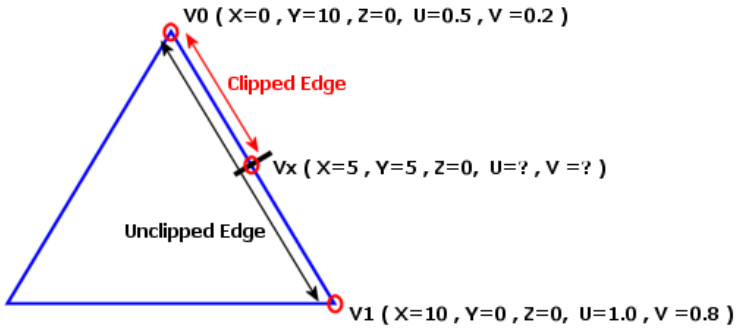
When this function returns, the ***vecIntersection*** variable will contain the x,y,z coordinates in world space for the new vertex of our clipped water quad. We build our final vertex using this position value and the correct color and alpha values:

```
// This is our new point
Points[PointCount].x = vecIntersection.x;
Points[PointCount].y = vecIntersection.y;
Points[PointCount].z = vecIntersection.z;
Points[PointCount].Diffuse = 0xBFFFFFFF;
```

At this point we will need to address what to do about the texture coordinates. As you might suspect, texture coordinates will also need to be clipped as well. For example if the first vertex had UV coordinates (0.5, 0.0) and the second had UV coordinates of (1.5, 1.0) and the plane intersected the

edge exactly halfway through the length of the edge, then the texture coordinates would have to be interpolated to find the new UV coordinates -- which would be (1.0 , 0.5) in this case.

Clipping texture coordinates is very similar to clipping line vertices. In this case, it is actually very simple to do because we have the original edge and the new edge. All we have to do is calculate the length of the original unclipped edge and the length of the new clipped edge. The unclipped edge is a vector created using v1 – v0 and the clipped edge vector can be created by doing Vx – v0 where Vx is the new vertex that we have just created which is positioned somewhere along the vector v1-v0. Once we have the unclipped and clipped edge vectors we can calculate their lengths. By dividing the length of the clipped edge by the length of the unclipped edge, we can determine how far along the original edge the intersection happened as percentage between 0.0 and 1.0.



The image above shows the edge of a triangle that has intersected a plane and has had the new vertex inserted on the plane (Vx). New UV coordinates must now be generated.

The first thing we do is calculate the length of the unclipped edge and the length of the clipped edge.

UnclippedEdgeVector = V1 – V0 = (10 , -10 , 0)
ClippedEdgeVector = Vx - V0 = (5 , -5 , 0)

Next we calculate the length of each vector. In this case the results are:

UnclippedEdgeLength = 14.142136
ClippedEdgeLength = 7.071068

If we divide the clipped edge length by the unclipped edge length we get a value between 0.0 and 1.0 describing how far along the unclipped edge the new vertex was inserted.

Percent = ClippedEdgeLength / UnclippedEdgeLength
Percent = 7.071068 / 14.142136 = 0.5

We can visually see that the result is correct because the inserted vertex is indeed halfway along the unclipped edge vector in the above diagram.

So in this example we know that the new vertex is halfway along the edge. We also know that the texture coordinates stored at each of the original vertices form an edge in texture space too. Therefore, we can create a 2D vector using the texture coordinates in the unclipped edge and scale it by the percent (0.5 in this example) to offset it from the coordinates in the edge:

Vx.tu = v0.tu + ((v1.tu – v0.tu) * Percent)
Vx.tv = v0.tv + ((v1.tv – v1.tv) * Percent)

Using the values from the above diagram:

Vx.tu = 0.5 + ((1.0 – 0.5) * 0.5) = 0.5 + 0.25 = 0.75
Vx.tv = 0.2 + ((0.8 – 0.2) * 0.5) = 0.2 + 0.3 = 0.5

The code that calculates the texture coordinates for the new vertex follows.

```
   // Calculate the texture coordinates.
   float LineLength = D3DXVec3Length( &((D3DXVECTOR3&)vecWaterPoints[v2] -
                                        D3DXVECTOR3&)vecWaterPoints[v1]) );

   float Distance = D3DXVec3Length(&(vecIntersection - (D3DXVECTOR3&)vecWaterPoints[v1]));

   float dist_len = Distance / LineLength;

   Points[PointCount].tu =
     vecWaterPoints[v1].tu + ((vecWaterPoints[v2].tu-vecWaterPoints[v1].tu)* dist_len);

   Points[PointCount].tv =
     vecWaterPoints[v1].tv + ((vecWaterPoints[v2].tv - vecWaterPoints[v1].tv) * dist_len);

   PointCount++;

  } // Next Vertex

} // End if Clip water
```

This is it for the clipped case. We do this for each vertex in the quad that requires clipping and at the end of the loop we have a new polygon stored in the Points[] array. We will render this new polygon in a moment.

If the quad does not need to be clipped then we can build the unclipped quad into the Points[] array. The quad will be rendered in its entirety because it cannot possibly intersect the near plane.

```
else
{
       Points[PointCount++] = CLitVertex(0.0f, WaterHeight, 0.0f, 0xBFFFFFFF, 0.0f, 0.0f);

       Points[PointCount++] = CLitVertex(0.0f,WaterHeight,m_nHeightMapHeight*m_vecScale.z,
                                         0xBFFFFFFF, 0.0f, 1.0f );

       Points[ PointCount++ ] = CLitVertex( m_nHeightMapWidth * m_vecScale.x, WaterHeight,
                                            m_nHeightMapHeight * m_vecScale.z,
                                            0xAFFFFFFF, 1.0f, 1.0f );
```

```
        Points[ PointCount++ ] = CLitVertex( m_nHeightMapWidth * m_vecScale.x, WaterHeight,
                                             0.0f, 0xBFFFFFFF, 1.0f, 0.0f );
}
```

To render the water polygon we need only one texture stage (stage 0). However in this demo (if the device supported single pass multi texturing), the second texture stage will currently contain the terrain detail texture. We will need to disable this texture stage to render the water plane because we want the alpha and RGB output from stage 0 to be used directly by the rasterizer.

```
    // Disable second texture stage if in use
    if(m_bSinglePass)
        m_pD3DDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_DISABLE );
```

If the Points[] array has less than three vertices in it, then it means the water polygon has been completely clipped by the near plane and we do not need to render anything.

```
    if ( PointCount > 2 )
    {
```

We set the alpha pipeline of stage 0 to extract the alpha value from the vertices. The color operation in stage 0 simply uses the RGB value sampled from the texture.

```
       // Setup alpha states and RGB states for rendering water
        m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_DIFFUSE );
        m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1 );

        m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
        m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
```

Next we enable alpha blending with the frame buffer and configure the source and destination blend render states so that the alpha output from the stage is used to mix the RGB color sampled from the water texture with the current contents of the frame buffer. This will provide a blue tint to the terrain pixels that can be seen through the water polygon.

```
        m_pD3DDevice->SetRenderState( D3DRS_SRCBLEND , D3DBLEND_SRCALPHA );
        m_pD3DDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
        m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
```

We set the water texture in stage 0 and inform the device that we will be using pre-lit vertices be setting the FVF.

```
        // Set our water texture into stage 0
        m_pD3DDevice->SetTexture( 0, m_pWaterTexture );

        // Set the FVF code for the water mesh.
        m_pD3DDevice->SetFVF( LITVERTEX_FVF );
```

Unlike normal polygons that we wish to have back face culled, we want the water polygon to be rendered from the front and the back. If we left back face culling enabled when the camera went

underneath the water the water polygon would not be seen from underneath and the surface would disappear once we were underneath it. Instead we prefer to be able to see the water surface from beneath as shown next.

To disable back face culling, we simply set the appropriate render state.

```
        // Disable back face culling (so we see it from both sides)
        m_pD3DDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_NONE );
```

We render the water polygon as a triangle fan using the DrawPrimitiveUP function (see Chapter Two). We pass in the number of primitives we wish to draw which (remember for a fan this is NumberOfVerts-2), and we pass in the Points[] array containing the vertices. The last parameter informs the device about the size of each vertex structure in bytes so that it can quickly move from one vertex to the next. Note that we could also reorder the vertices if we wanted to keep backface culling on, but it is nice to know how to turn it off and on, just in case.

```
        // Render polygon
        m_pD3DDevice->DrawPrimitiveUP(D3DPT_TRIANGLEFAN,
                                      PointCount - 2, Points, sizeof(CLitVertex));
```

Finally we disable alpha blending and re-enable back face counter-clockwise culling so that the terrain is rendered correctly in the next frame.

```
        // Reset states
        m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );
        m_pD3DDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CCW );

    } // if pointcount>2
```

If the camera is in the water (intersecting or completely underneath), we will need to render our screen effect. We call the CCamera::RenderScreenEffect function to draw the T&L alpha blended quad. This function will be covered in a moment.

```
    // Render alpha blended quad if we are underwater
    if ( pCamera && (pCamera->GetPosition().y - 10.0f) < WaterHeight )
    {
```

```
        pCamera->RenderScreenEffect(m_pD3DDevice,CCamera::EFFECT_WATER,
                                    *(ULONG*)(&WaterHeight) );

    } // End if render water effect
```

We re-enable the color operations in the second texture stage again if the device is using single pass multi- texturing because we disabled it to render the water polygon. If we did not do this, the next time the terrain was rendered the detail texture would be missing. Note that this render state inter-dependency can be prone to bugs, so we will introduce a system in the next course to allow for proper state management across scene objects.

```
    // Re-enable second texture stage if required
    if ( m_bSinglePass )
         m_pD3DDevice->SetTextureStageState(1, D3DTSS_COLOROP, GetGameApp()->GetColorOp());
}
```

The full code listing for this function follows for easier reading:

```
void CTerrain::RenderWater( CCamera * pCamera )
{
    CLitVertex  Points[5];
    int         PointCount = 0;

    // Retrieve floating point water height
    float WaterHeight = WaterLevel * m_vecScale.y;

    // If we are close enough to the water, we need to clip
    if ( pCamera && (pCamera->GetPosition().y - 10.0f) < WaterHeight )
    {
        // Build a combined projection / view matrix
        D3DXMATRIX mtxCombined = pCamera->GetViewMatrix() * pCamera->GetProjMatrix();

        // Extract the near clipping plane.
        D3DXPLANE NearPlane;
        NearPlane.a = -(mtxCombined._13);
        NearPlane.b = -(mtxCombined._23);
        NearPlane.c = -(mtxCombined._33);
        NearPlane.d = -(mtxCombined._43);
        D3DXPlaneNormalize( &NearPlane, &NearPlane );

        // Build initial 4 corner vectors
        CLitVertex vecWaterPoints[4];
        vecWaterPoints[0] = CLitVertex(0.0f, WaterHeight, 0.0f, 0xBFFFFFFF, 0.0f, 0.0f);
        vecWaterPoints[1] = CLitVertex(0.0f,WaterHeight,
                                       m_nHeightMapHeight*m_vecScale.z,
                                       0xBFFFFFFF, 0.0f, 1.0f );
        vecWaterPoints[2] = CLitVertex(m_nHeightMapWidth * m_vecScale.x, WaterHeight,
                                       m_nHeightMapHeight * m_vecScale.z,
                                       0xAFFFFFFF, 1.0f, 1.0f );
        vecWaterPoints[3] = CLitVertex(m_nHeightMapWidth * m_vecScale.x, WaterHeight,
                                       0.0f, 0xBFFFFFFF, 1.0f, 0.0f );

        // Clip this quad against the plane, discard anything in front
        for ( int v1 = 0; v1 < 4; v1++ )
        {
```

```
            int v2 = (v1 + 1) % 4;

            // Classify each point in the edge
            int Location1 = 0, Location2 = 0;

            float result = D3DXPlaneDotCoord(&NearPlane,(D3DXVECTOR3*)&vecWaterPoints[v1]);
            if ( result < -1e-5f ) Location1 = -1; // Behind
            if ( result >  1e-5f ) Location1 =  1; // In Front

            // Keep it if it's on plane
            if ( Location1 == 0 )
            {
                Points[ PointCount++ ] = vecWaterPoints[v1];
                continue; // Skip to next vertex
            }

            result = D3DXPlaneDotCoord( &NearPlane, (D3DXVECTOR3*)&vecWaterPoints[v2] );
            if ( result < -1e-5f ) Location2 = -1; // Behind
            if ( result >  1e-5f ) Location2 =  1; // In Front

            // If its not in front, keep it.
            if ( Location1 != 1 ) Points[ PointCount++ ] = vecWaterPoints[v1];

            // If the next vertex is not causing us to span the plane then continue
            if ( Location2 == 0 || Location2 == Location1 ) continue;

            // Calculate the intersection point
            D3DXVECTOR3 vecIntersection;
            D3DXPlaneIntersectLine( &vecIntersection, &NearPlane,
                                    (D3DXVECTOR3*)&vecWaterPoints[v1],
                                    (D3DXVECTOR3*)&vecWaterPoints[v2] );

            // This is our new point
            Points[PointCount].x = vecIntersection.x;
            Points[PointCount].y = vecIntersection.y;
            Points[PointCount].z = vecIntersection.z;
            Points[PointCount].Diffuse = 0xBFFFFFFF;

            // Calculate the texture coordinates.
            float LineLength = D3DXVec3Length( &((D3DXVECTOR3&)vecWaterPoints[v2] -
                                               (D3DXVECTOR3&)vecWaterPoints[v1]) );

            float Distance = \
                     D3DXVec3Length(&(vecIntersection-(D3DXVECTOR3&)vecWaterPoints[v1]));

            Points[PointCount].tu=vecWaterPoints[v1].tu+
                ((vecWaterPoints[v2].tu-vecWaterPoints[v1].tu) * (Distance / LineLength));

            Points[PointCount].tv=vecWaterPoints[v1].tv+((vecWaterPoints[v2].tv-
                                  vecWaterPoints[v1].tv) * (Distance / LineLength));
            PointCount++;

        } // Next Vertex

    } // End if Clip water
    else
    {
        Points[PointCount++] = CLitVertex(0.0f, WaterHeight, 0.0f, 0xBFFFFFFF, 0.0f, 0.0f);
        Points[PointCount++] = CLitVertex(0.0f,  WaterHeight,
```

```
                                        m_nHeightMapHeight * m_vecScale.z,
                                        0xBFFFFFFF, 0.0f, 1.0f );
        Points[PointCount++] = CLitVertex(m_nHeightMapWidth * m_vecScale.x, WaterHeight,
                                        m_nHeightMapHeight*m_vecScale.z,
                                        0xBFFFFFFF,1.0f,1.0f);
        Points[PointCount++] = CLitVertex(m_nHeightMapWidth * m_vecScale.x, WaterHeight,
                                        0.0f, 0xBFFFFFFF, 1.0f, 0.0f );
    } // End if Just Build unclipped

    // Disable second texture stage if in use
    if ( m_bSinglePass )
         m_pD3DDevice->SetTextureStageState( 1, D3DTSS_COLOROP, D3DTOP_DISABLE );

    if ( PointCount > 2 )
    {
        // Setup alpha states for rendering water
        m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_DIFFUSE );
        m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1 );
        m_pD3DDevice->SetRenderState( D3DRS_SRCBLEND , D3DBLEND_SRCALPHA );
        m_pD3DDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
        m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );

        // Set our water texture into stage 0
        m_pD3DDevice->SetTexture( 0, m_pWaterTexture );

        // Set the FVF code for the water mesh.
        m_pD3DDevice->SetFVF( LITVERTEX_FVF );

        // Disable back face culling (so we see it from both sides)
        m_pD3DDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_NONE );

        // Render polygon
        m_pD3DDevice->DrawPrimitiveUP(D3DPT_TRIANGLEFAN,
                                      PointCount - 2, Points, sizeof(CLitVertex));

        // Reset states
        m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );
        m_pD3DDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CCW );
     }

    // Render alpha blended quad screen effect if we are underwater
    if ( pCamera && (pCamera->GetPosition().y - 10.0f) < WaterHeight )
    {
        pCamera->RenderScreenEffect(m_pD3DDevice, CCamera::EFFECT_WATER,
                                    *(ULONG*)(&WaterHeight) );
     }

    // Re-enable second texture stage if required
    if ( m_bSinglePass )
         m_pD3DDevice->SetTextureStageState(1, D3DTSS_COLOROP, GetGameApp()->GetColorOp());
}
```

Rendering the water quad was certainly a bit more complicated than we might have first imagined. However, we did manage to learn some important new concepts in trying to address the problems. For example, we now know how to clip polygons to planes (view frustum or otherwise). This is a technique that will come in handy down the road when we study spatial partitioning and advanced data structures. We also learned how find the intersection point of a ray with a plane. This can also prove to

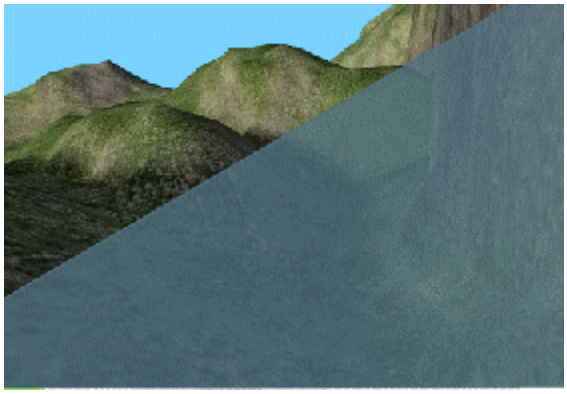
be useful later on when we need to develop collision systems. So while we had to go down a slightly bumpy road, the journey was worth the effort.

Note that many games forbid you from doing having a partially submerged camera by automatically pushing the player all the way under as soon as they enter the water. If you decided to employ such a strategy as well, you could remove the clipping code from the above function -- reducing the code to a mere fraction of its current size. It is nice however to know how to handle the transition if you wish to do it the way we did.

Of course, we are not quite done with our water rendering. We still need to examine the code that handles the underwater effect.

CCamera::RenderScreenEffect

This function renders an alpha blended polygon over the section of the frame buffer which is perceived to be underneath the water plane. Because the camera may be rolled or pitched to any arbitrary angle this means we will have to also clip this 2D polygon to get the correct slope of the horizon on the top edge of the quad. We want it to align with the water polygon just rendered in the last function.



The image on the left shows the effect in action. Although it is hard to see in the image, we are half in and half out of the water. The actual water quad would be seen as a thin blue line going diagonally across the screen because we are looking at a cross section of an extremely thin polygon. What the RenderScreenEffect function will do is fill all the area in the frame buffer that is underneath the water line with a blue polygon (alpha blended). This transforms the water from being perceived as a thin polygon to looking and feeling like a volume of water. This involves calculating the water line that the water polygon forms on the near plane, building a screen space quad, and clipping it to this line. We then alpha blend the quad with the frame buffer and we are done. It certainly sounds easy enough to do but there is a little more to it than you might expect, so we will step through the code a section at a time.

```
void CCamera::RenderScreenEffect( LPDIRECT3DDEVICE9 pD3DDevice, SCREEN_EFFECT Effect,
                                  ULONG Value )
{
    CTLitVertex Points[5];
    int         PointCount = 0;

    pD3DDevice->SetRenderState( D3DRS_ZENABLE, D3DZB_FALSE );
```

The function takes a pointer to a device, a **SCREEN_EFFECT** parameter, and a value that describes the height of the water. The reason these parameters sound generic is so that we can implement different effects in the future. The second parameter is of type **SCREEN_EFFECT** which is defined in CCamera.h

as an enumerated type. Currently the only member of the enumerated type is **`EFFECT_WATER`** which is the value passed into function by the CTerrain::RenderWater function. The third parameter is passed the water height value in world space from the CTerrain::RenderFunction.

The first thing we do is allocate an array of 5 pre-transformed and pre-lit vertices (screen space vertices). There are 5 points because the screen space quad may have to be clipped to emulate the water line of the water quad and as we discovered in the previous function, clipping a quad to a plane can introduce an additional vertex.

We disabled the Z-Buffer by setting the render state (D3DRS_ZENABLE) to D3DZB_FALSE. While we could have just disabled z-writing, there is little point in performing all of the extra per-pixel depth tests in this case since we know that nothing will occlude this polygon. Also note that it will be the last thing we render in our scene so that nothing will overdraw the water pixels either.

Next we check the value of the SCREEN_EFFECT parameter passed in to make sure that it is EFFECT_WATER (currently the only defined screen effect in our camera class). If you decide to add new effects to the camera class yourself, you can add additional cases to the following switch statement. The first thing we do is extract the near plane from the projection matrix as we did in the previous function.

```
    switch ( Effect )
    {
        case EFFECT_WATER:
        {
            D3DXVECTOR3 vecPoint[2], PlaneNormal;
            float       DistFromPlane, LineLength;

            // Retrieve floating point water height
            float WaterHeight = *(float*)(&Value);

            // Build a combined projection / view matrix
            D3DXMATRIX mtxCombined = m_mtxView * m_mtxProj;

            // Extract the near clipping plane.
            D3DXPLANE NearPlane;
            NearPlane.a = -(mtxCombined._13);
            NearPlane.b = -(mtxCombined._23);
            NearPlane.c = -(mtxCombined._33);
            NearPlane.d = -(mtxCombined._43);
            D3DXPlaneNormalize( &NearPlane, &NearPlane );

            // Store plane normal for easy access later on.
            PlaneNormal = D3DXVECTOR3( NearPlane.a, NearPlane.b, NearPlane.c );
```

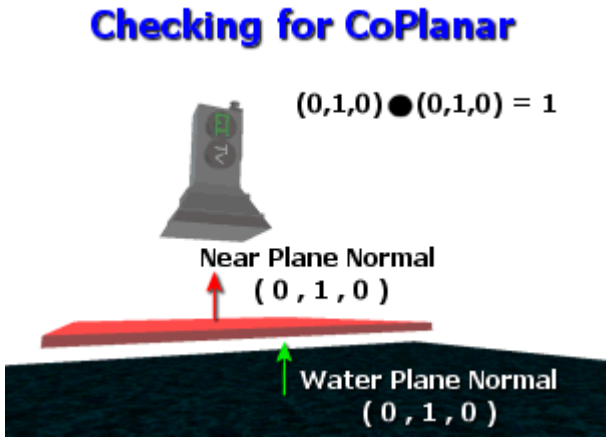
It is now time to create our screen effect polygon which may or may not need to be clipped depending on the position and orientation of the camera. If the near plane is totally or partially in the water then a screen effect polygon will be needed.

First there is the case where a quad covering the entire frame buffer is needed. This happens when the camera is completely submerged. We compute the dot product between the near plane normal and the

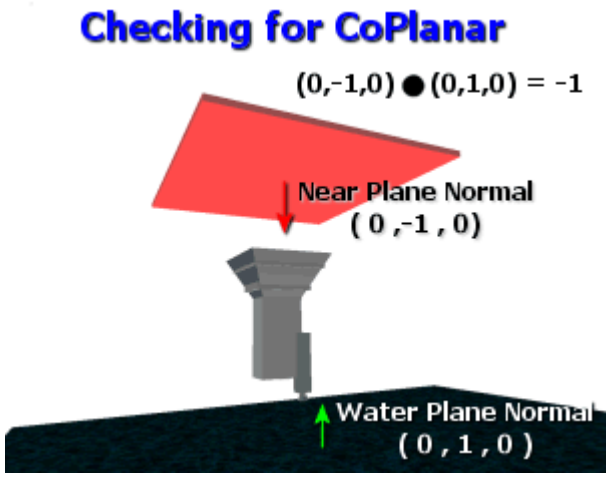
water polygon normal (0,1,0) to see if the near plane and the water polygon are coplanar. There are two cases when the near plane will be coplanar:

1. when the camera is looking directly down at the water
2. when the camera is looking directly away from the water up into the sky

In the first case where the camera is looking down at the water, the two normals will be the same and the dot product will return 1.0 (approximately) as shown next:



The second case has the camera facing directly away from the water and the dot product of the two vectors would return approximately –1.0. (We say 'approximately' because we will use an epsilon value of 0.0001 for such comparisons to provide tolerance for floating point inaccuracies). Below you can see the second coplanar case where the camera is facing away from the water; the near plane normal and the water normal have opposing directions.



So the first thing we will do is check if the dot product between the two vectors is either –1 or 1 by getting the absolute value of the dot product and subtracting from 1.0. If the value is approximately 0.0 then the camera is either facing the water or facing away from the water directly. If either is true, then we know that the screen effect quad will not need to be clipped, even if the camera is in the water. You

can see that in the first of the two coplanar images above, even if we nudged the near plane down so that it is in the water, it will be either under the water or not -- but it cannot be partially in the water. We use an epsilon of 1e-3f (0.0003) for checking zero with tolerance.

```
            // If the near plane and water plane are 'almost' coplanar
            // then we will have problems, so we should test for this case
            float fDot = fabsf(D3DXVec3Dot( &PlaneNormal, &D3DXVECTOR3( 0, 1, 0 ) ));

            if ( (1.0f - fDot) < 1e-3f )
            {
```

Because the image is rendered on the near plane we must check that it is not in the water. Note that this is different than checking the camera position itself, which is always slightly behind the near plane. We simply calculate the distance from the camera position to the near plane (using the D3DXPlaneDotCoord function) and slide the camera position backwards along the near plane normal. We now have a point that is on the plane. We can check whether this point is below the water line or not. Note that in the following code we only project the y component of the camera position onto the near plane. This is because we only need to know if the y value of the projected point is above or below the water line.

```
                // Project a point, from the camera pos out to the near plane itself
                // (we only compute the y component here, because we only need the height)
                float Point = \
                      m_vecPos.y+(-PlaneNormal.y*D3DXPlaneDotCoord(&NearPlane,&m_vecPos ));
```

Once we have the height of the near plane in world space we to check whether it is under the water or not. If not, we can break from the EFFECT_WATER case because there is nothing to do.

```
                // If this point is above the water line, then bail
                if ( Point > WaterHeight ) break;
```

When the near plane is underneath water, we need to render a large blue transparent quad over the frame buffer. We are going to use T&L vertices to define the quad in screen space coordinates. So we build the four vertices of our quad and store them in the 'Points[]' array where they will be rendered from later in the function. The four vertices have X and Y values that map to the four corners of the frame buffer and all have Z-Buffer values of 0.0. The fourth value is the RHW value which we described as being 1/view space z. We set this value to 1.0 as discussed. Finally we set the color of each vertex to 0xBF547686 which is color ARGB (191, 84, 118, 134). This gives us a murky bluish/green color with ¼ transparency.

```
                // We should render, so simply build the 4 points
                Points[PointCount++]=CTLitVertex((float)m_Viewport.X,(float)m_Viewport.Y,
                                                 0.0f, 1.0f, 0xBF547686);

                Points[PointCount++]=CTLitVertex((float)m_Viewport.X + m_Viewport.Width,
                                                (float)m_Viewport.Y,
                                                 0.0f, 1.0f, 0xBF547686 );

                Points[PointCount++]=CTLitVertex((float)m_Viewport.X+m_Viewport.Width,
```

```
                                                (float)m_Viewport.Y+m_Viewport.Height,
                                                 0.0f,1.0f, 0xBF547686);

               Points[PointCount++]=CTLitVertex((float)m_Viewport.X,
                                                (float)m_Viewport.Y + m_Viewport.Height,
                                                 0.0f, 1.0f, 0xBF547686);
          } // End if coplanar
```

The coplanar case is nice and easy because we either render a quad or not depending on the height of the camera. If the near plane and water polygon are not coplanar however then it means the camera may be partially submerged and possibly even rolled such that the water line travels diagonally across the screen. We will need to calculate this water line and clip our screen effect quad so that it reflects this water line properly.

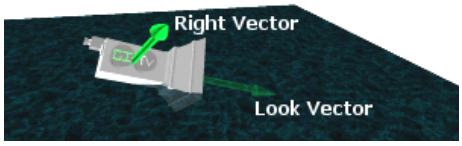
```
          else
          {
               D3DXVECTOR3 vecIntersect[2], vecRight, vecOut;
```

To calculate the slope of the water line we use the camera right vector and flatten it onto the water plane. We make sure to still keep it aligned with the camera actual right vector. We do this by performing a cross product between the camera look vector and the water plane's normal. This will return a unit vector which is perpendicular to the two input vectors. The code to calculate this vector is shown below.

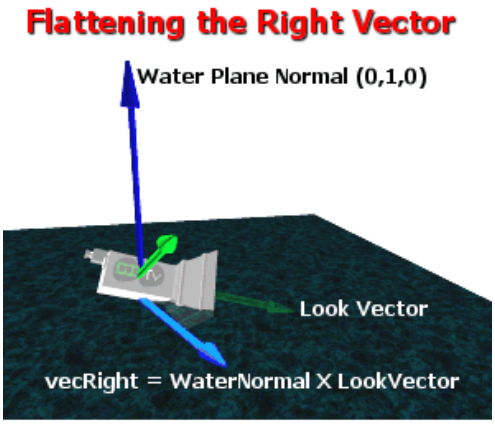
```
               D3DXVec3Cross( &vecRight, &D3DXVECTOR3( 0, 1, 0), &m_vecLook );
```

The image below shows a camera that is half in and half out of the water, intersecting the water at a complex angle. The diagram also depicts the orientation of the camera look and right vectors at this point.

Camera Partially in Water



The next image shows what this new flattened right vector (vecRight) would look like.

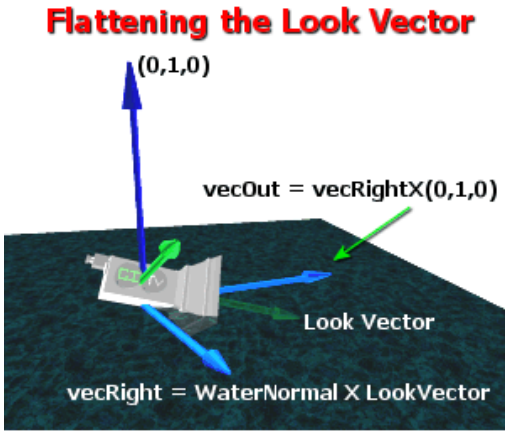


If we converted this vector into camera space, it would describe the slope of the water on the screen. Try it for yourself in your head. Imagine rotating the camera in the image above so that it is facing perfectly to the right on the page. Also imagine that you rotated all the vectors and the water plane by the same amount when rotating the camera. The flattened right vector and the water polygon would now be sloping downwards. The camera right and the flattened right vector would thus describe the direction of the slope the water makes on the monitor screen. We will see this being used in a moment.

Next we will need another vector which describes the orientation the camera is facing along the water polygon plane. You can think of this as being the camera look vector flattened onto the water plane. We perform the cross product on the water polygon normal (0,1,0) and the flattened right vector we just calculated to produce the new vector (vecOut).

```
D3DXVec3Cross( &vecOut, &m_vecRight, &D3DXVECTOR3( 0, 1, 0) );
```

The image below shows us what this new vector would look like:

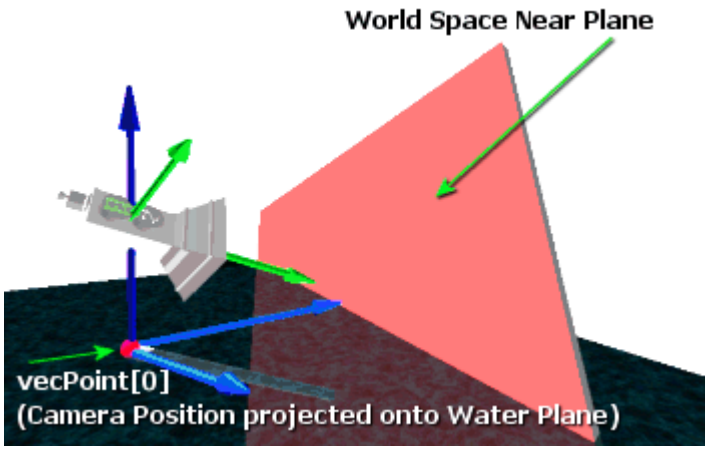


We now have new right and look vectors projected onto the water polygon plane. These vectors remain (partially) aligned to the camera look and up vectors. We will see why we need these vectors in a moment.

Next we need to project the camera position onto the water plane. To do this we set the Y component of the camera position to the water height in world space. This provides a world space point that is on the water polygon and directly underneath the camera.

```
// Project a point (vecPoint[0]) onto the near plane, at the water level.
vecPoint[0] = D3DXVECTOR3( m_vecPos.x, WaterHeight, m_vecPos.z );
```

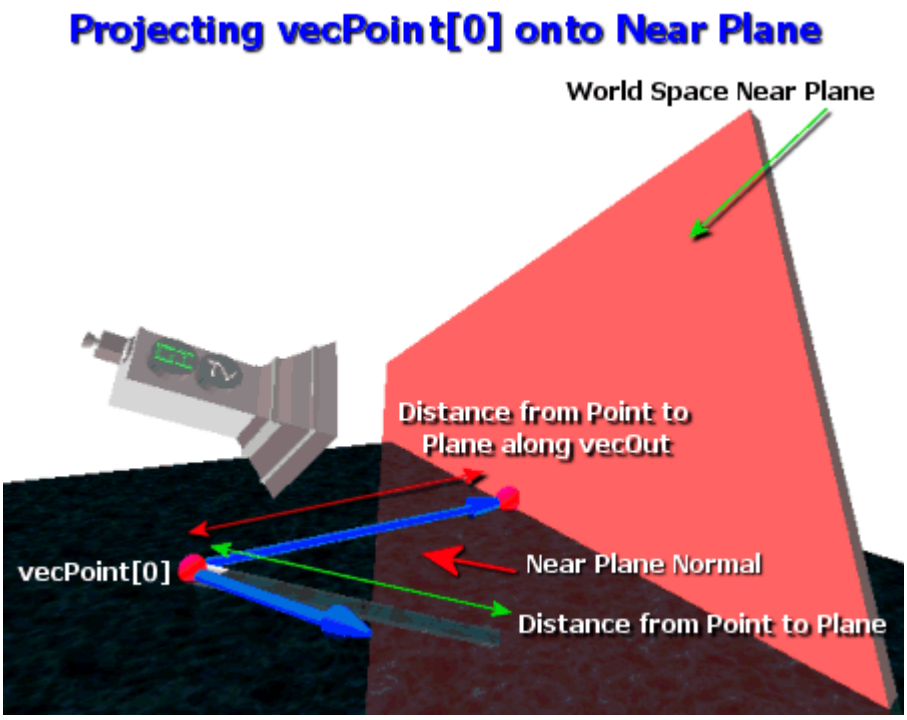
If this is hard to imagine, the following image should help. We have taken some liberties with the diagram so that we can see things more clearly. The camera is nudged up just out of the water purely for demonstration purposes. In reality, the camera would be much closer to the water plane.



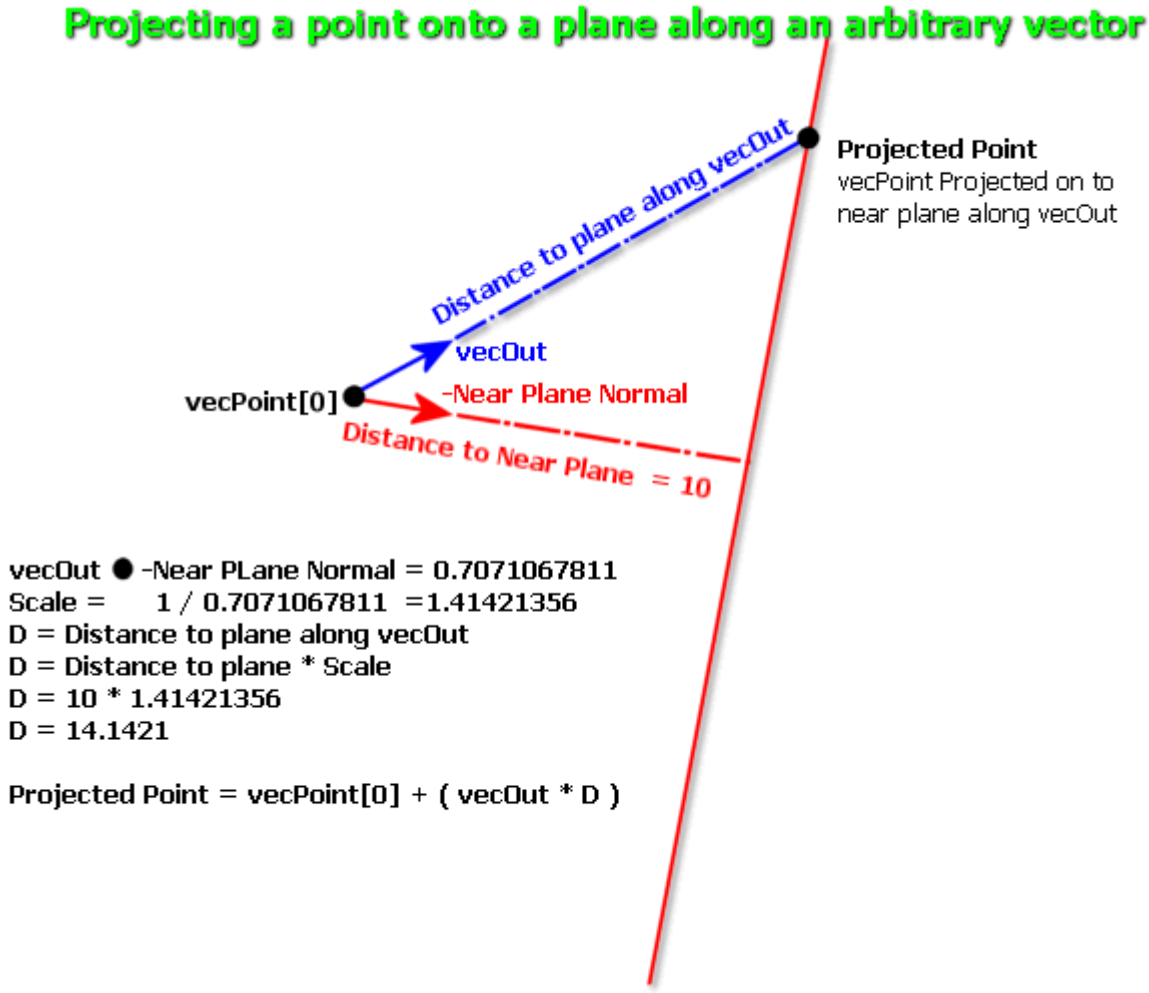
The projected point which was stored in the variable 'vecPoint[0]' now sits nicely on the water at the center of the axes formed by the flattened look and right vectors and the water plane normal.

We also see a representation of the world space near plane positioned in front of the camera. You can think of it as the frame buffer converted into world space. Everything that is rendered is always rendered on the near plane so this is not a really bad analogy. Although planes are infinite, we see only a section of the near plane. This can be thought of as the projection window on the near plane. Recall from Chapter One that any projection coordinates that end up in the range of –1.0 to 1.0 are considered to be inside the projection window and rendered to the screen.

Our next job is to take this point on the water and project it along the flattened look vector such that it sits on the near plane at its center.



The above image shows us that in order to do this we need to find the distance from our point to the near plane along the flattened look vector. We know that to get the distance from a point to a plane we can use the D3DXPlaneDotCoord function. However the problem with this function is that it provides the shortest distance from the point to the plane along the plane normal as shown by the green arrow in the above diagram. But in our case we need to calculate the distance along the flattened look vector, not the plane normal, as shown by the red arrow above. The solution is easy enough. We first calculate the distance to the plane using the D3DXPlaneDotCoord function. This returns the distance to the plane from the point along the plane normal. Once we have this distance to the plane we can perform the dot product on the flattened look vector and the reversed plane normal. This will return the cosine of the angle between the two vectors. We can then use this to scale the distance to the plane such that it describes the distance to the plane along the flattened look vector. This technique of calculating the distance to a plane along an arbitrary vector is shown below.

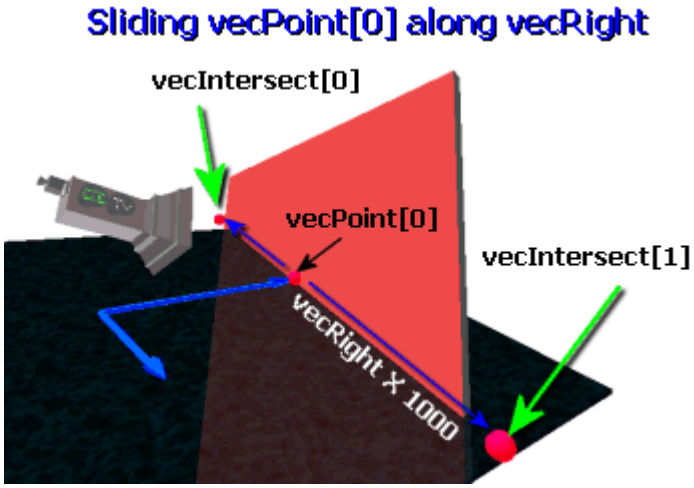


In this example we used the D3DXPlaneDotCoord function to return the distance from the point (vecPoint[0]) to the plane. The result was a distance of 10 units. This is how far we would need to move vecPoint[0] along the red dotted line for it to sit on the plane. Next we flip the sign of the plane normal so that both vectors are pointing in the same direction. A dot product between the unit length flattened look vector (vecOut) and the unit length reverse plane normal is performed. It returns the cosine of the angle between the vectors. Because the cosine will be 1.0 when the vectors are the same and some smaller value between 0.0 and 1.0 when there is any angle between them, this is the opposite of what we want. As the angle increases between the vectors you can see that the length of the blue dotted line in the above diagram would actually increase. This means we would need to push the point a larger distance along this line for it to hit the plane. As the angle grows between the vectors, the cosine of the angle becomes smaller. Scaling the distance with this value would make it smaller and not larger as it should. This is easy enough to remedy. We simply multiply distance by the inverse (1/CosineOfAngle). In this case, as the angle between the vectors becomes larger, the distance is scaled by a greater amount. Once we have the projected distance to the plane, we can simply move vecPoint[0] along the flattened look vector (vecOut) by this amount. The code is shown below.

```
// Project the water point onto the plane along the vecOut
DistFromPlane  = D3DXPlaneDotCoord( &NearPlane, &vecPoint[0] );
DistFromPlane *= (1 / D3DXVec3Dot( &(-PlaneNormal), &vecOut ));

// Shift the point forward so that it sits on the plane.
vecPoint[0] += vecOut * DistFromPlane;
```

We now have a point that is centered on the world space near plane at the height of the water. The flattened right vector we calculated earlier describes the direction to the right (and left if we negate this vector) of the camera aligned with the water plane. What we will do now is create two extreme points off to the far right and far left of the camera, whilst remaining on the water plane.

```
//Shift this projected point to the left and right
//to get our two intersection points
vecIntersect[0] = vecPoint[0] - (vecRight * 1000.0f);
vecIntersect[1] = vecPoint[0] + (vecRight * 1000.0f);
```



The two vecIntersect[] points are calculated by taking the near plane center point on the water line that we just calculated and shunting it forwards and backwards along the flattened right vector. The exact distance we shunt does not really matter as long as it is a large enough amount to ensure that when the points are converted into projection space, they lay well outside the projection window on each side. The image on the left shows what vecIntersect[0] and vecIntersect[1] now look like in world space. The image is not to scale since the two points would actually be projected out much further to the left and right of the camera. All that really matters is that these points are moved outside the camera FOV. Another very important point is that these points are still on the water polygon plane. When these points are converted into projection space (and later screen space) they will define the two end points of a line. This line defines the slope of the water in screen space.

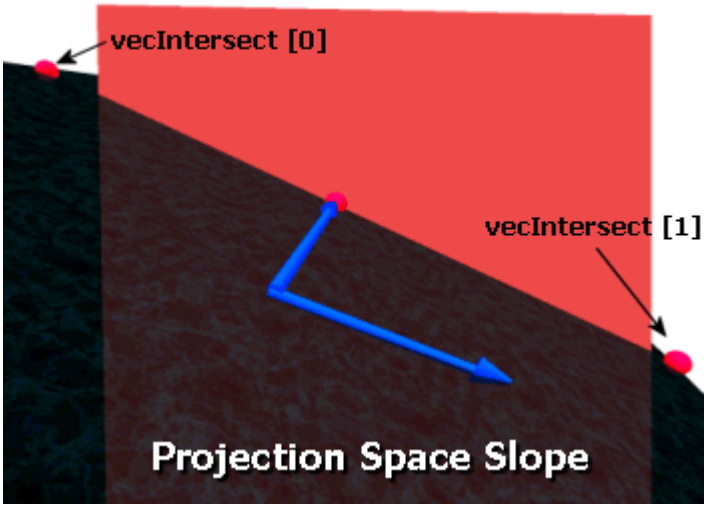
Our next job is to convert these world space end points into view space and then into projection space. We do this by multiplying the points with the view matrix and the projection matrix. We already have a combined view/projection matrix available which we used to extract the world space view plane, so let us multiply the two end points (vecIntersect[0] and vecIntersect[1]) by this combined matrix. This will convert the vertices into projection space. Points in the –1 to +1 range on both the X and Y axis are considered to be inside the projection window and within the camera FOV.

```
// Project the two intersection points into 'Projection' space
D3DXVec3TransformCoord( &vecPoint[0], &vecIntersect[0], &mtxCombined );
D3DXVec3TransformCoord( &vecPoint[1], &vecIntersect[1], &mtxCombined );
```

The projection space water line end points are stored in vecPoint[0] and vecPoint[1]. As these are in projection space, we may as well define our initial screen quad in projection space too and perform the clipping there. Once we have the clipped projection space polygon, we can convert the projection space vertices into screen space and render.

Recall from Chapter One that in projection space, the entire visible area of the screen can be defined with (X, Y) coordinates in the –1 to +1 range. Coordinate (0, 0) is at the center of the screen, coordinate (-1, -1) is the bottom left corner of the screen and coordinate (1, 1) is the top right corner of the screen. With this information we can quickly figure out how to build our projection space quad. It is simply a square where the top left corner vertex coordinate is (-1, -1) and the bottom right corner vertex is at coordinates (1, 1).

In order to clip a polygon, we need a plane to clip it to. Our water line describes the slope that the plane should have, but it is just a 2D line. However this works for us since our screen space quad is also a 2D shape. So we will do a simple line/polygon clipping routine that follows the same logic as our plane/polygon approach. In that sense we will treat this line as a '2D plane'. If we have a 2D normal for this line then we can clip the 2D quad to it just as we clipped the 3D quad to the plane in the previous function. As it happens, when we have a line described in 2D coordinates, generating the normal is quite straightforward:



Line = v0 – v1 (this line is described by our two projection space points)
EdgeNormal.x = - (v1.y – v0.y)
EdgeNormal.y = (v1.x – v0.x)

Let us imagine that we have a straight horizontal line consisting of v1 (0, 0) and v2 (10, 0). We know that because this line is a perfect horizontal, rotating it 90 degrees should give a perfect vertical. This vertical would describe the 'normal' of the line.

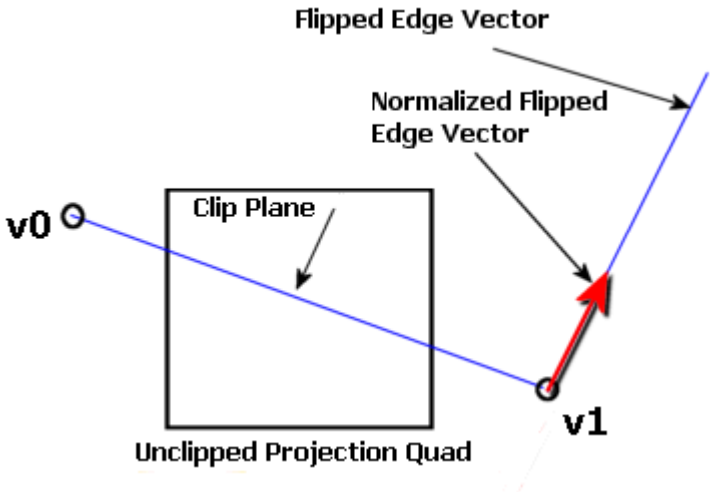
2DNormal.x = - (0 – 0) = 0
2DNormal.y = (10 - 0) = 10

Therefore we have generated a vector perpendicular to the line as shown below.

2DNormal = (0, 10)

All we have to do now is normalize the 2D vector so that it is unit length and we have a normal for our line. We can use it to classify the points of our unclipped screen effect quad while clipping.

In the following image we see the generation of an edge normal for the water line defined by points v0 and v1. Notice that once we normalize the flipped edge, we have a line cutting right across the projection window. This line will be used to clip our projection space quad.



The following code demonstrates generating the edge normal from our water line and creating the initial unclipped projection space quad.

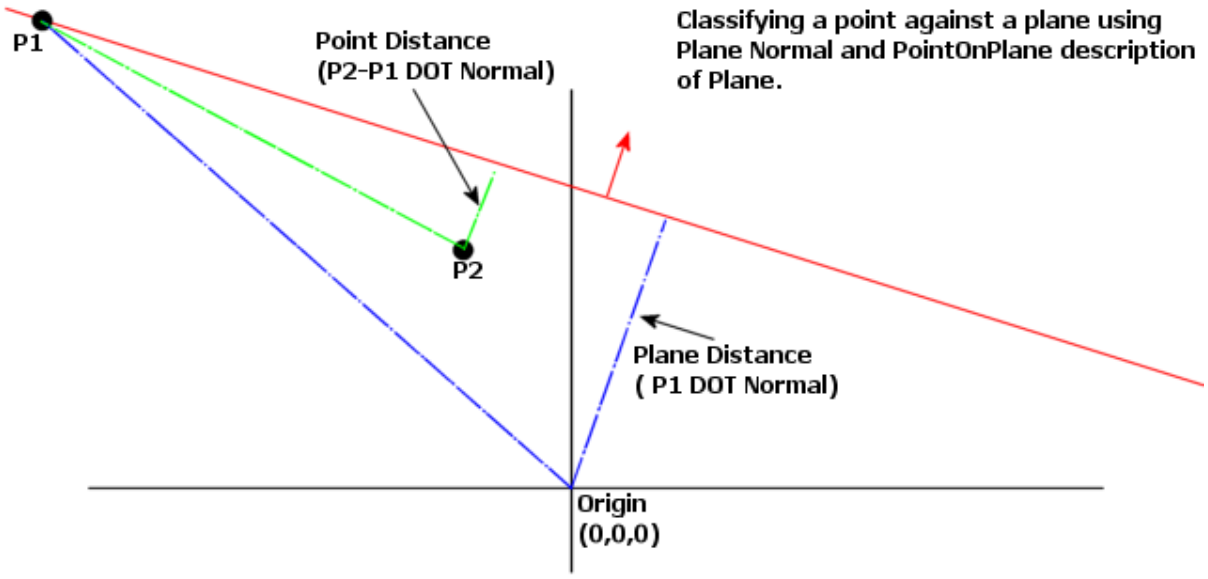
```
// Generate our 2d plane
D3DXVECTOR2 vecPointOnPlane = D3DXVECTOR2( vecPoint[1].x, vecPoint[1].y );
D3DXVECTOR2 vecPlaneNormal;
vecPlaneNormal.x = -(vecPoint[1].y - vecPoint[0].y);
vecPlaneNormal.y =  (vecPoint[1].x - vecPoint[0].x);
D3DXVec2Normalize( &vecPlaneNormal, &vecPlaneNormal );

// Build initial 4 corner vectors
D3DXVECTOR2 vecScreenPoints[4];
vecScreenPoints[0] = D3DXVECTOR2( -1.0f,  1.0f );
vecScreenPoints[1] = D3DXVECTOR2(  1.0f,  1.0f );
vecScreenPoints[2] = D3DXVECTOR2(  1.0f, -1.0f );
vecScreenPoints[3] = D3DXVECTOR2( -1.0f, -1.0f );
```

We are now just about ready to start clipping the quad to our new 2D water line. However we have not calculated the distance to the plane from the origin of the coordinate system. As you know already we need to define the plane equation to classify points against the plane. Although we could easily calculate the plane distance by performing the dot product on the plane normal we just calculated and any point know to be on the plane (any of our line edge points would do) this does present us with a nice opportunity to look an alternative way of storing a plane and classifying points against it.

Alternative Plane Method

Up to this point we have represented a plane using a plane normal and a value describing the distance from the origin of the coordinate system to the plane along the plane normal. This is the most common

plane representation and plugs nicely into the plane equation. However, we can also work with planes even if we do not have the distance value. All we need is the plane normal and a point that is known to be on the plane. This can be useful when we are assembling planes from polygons for collision detection because we usually know the polygon normal. Furthermore, because all vertices in the polygon are coplanar, any vertex in the polygon can be used as the point on the plane.



We already know that we can use a point on the plane to calculate a distance from the origin (see Chapter One). Take a look at the point P1 and the vector it forms from the origin (the blue dotted line on the left). We know that if we perform the dot product between this vector and the unit length plane normal we get the cosine of the angle between them scaled by the length of vector P1. In other words it is like taking the plane normal, placing it at the origin and scaling it such that it reaches the plane (the blue dotted line on the right). This gives us the distance to the plane. Now, it is because the plane normal is placed at the origin for the dot product that we have to add the distance of the plane onto the result. Otherwise the plane would be assumed to pass through the origin. In other words, when we use D3DXPlaneDotCoord, it temporarily moves the plane so that it passes through the origin, gets the cosine of the angle, and then adds the distance back on that had been removed by the dot product.

Classifying a point against a plane when we do not have the plane distance is very similar. In the above example you can see that we wish to classify the point P2 against the plane. If we subtract this point (vector) from the point known to be on the plane we effectively move point P2 to the origin and offset the point on the plane (therefore the plane itself) by the same amount. Because P2 is at the origin, the classification of the point has been simplified to merely calculating the plane distance -- just like we do when we normally calculate the distance of a plane from the origin. When we subtract vector P2 from P1 we get a non unit length vector from P2 to P1. When we perform the dot product with this vector and the unit length plane normal we get the length of vector (P2-P1) scaled by the cosine of the angle

between them. This is the length of the green dotted line on the right in the above diagram. Therefore, we could calculate the distance from P2 to the plane doing the following:

Distance = D3DXVec3Dot(&(P2-P1) , PlaneNormal)

The result tells us the distance to the plane. The sign of the result tells us whether the point is behind or in front of the plane just as this was the case when we used D3DXPlaneDotCoord.

Of course, if you do not like this method then you could just calculate the distance to the plane from the origin and then use the D3DXPlaneDotCoord function as before. In our code, we are dealing with only 2D vectors but the same still applies; we just perform 2D dot products instead.

Clipping the Screen Space Polygon

The next section of code will look familiar as it classifies each of the edges of our projection space quad against the water line plane just created. It clips the edges as we did in the previous function when clipping the water quad itself. In this section we use the PointOnPlane method of classifying the vertices against the plane. The normal of our water line is facing out of the water so we keep the section of the quad that is behind the plane and remove the rest. First we classify each vertex in the current edge to see whether they are in front or behind of the plane (above or below the water line respectively).

```
// Clip this quad against the plane, discard anything in-front
for ( int v1 = 0; v1 < 4; v1++ )
{
    int v2 = (v1 + 1) % 4;

    // Classify each point in the edge
    int Location1 = 0, Location2 = 0;

    floatresult=D3DXVec2Dot(&(vecScreenPoints[v1]-vecPointOnPlane),
                            &vecPlaneNormal);
    if ( result < -1e-5f ) Location1 = -1; // In Behind
    if ( result >  1e-5f ) Location1 =  1; // In Front

    result=D3DXVec2Dot(&(vecScreenPoints[v2] - vecPointOnPlane),
                       &vecPlaneNormal);
    if ( result < -1e-5f ) Location2 = -1; // Behind
    if ( result >  1e-5f ) Location2 =  1; // In Front
```

When the current vertex is below the water line we need to add it to our new list of screen space vertices. Since our quad is currently made of projection space vectors, we need to convert them into a screen space. Refer back to Chapter One if you do not remember how to do this. If the current vertex we are processing is on the plane itself then we add it to the vertex list of the clipped quad and skip to the next vertex.

```
    if ( Location1 == 0 )
    {
```

```
                    Points[PointCount].x = vecScreenPoints[v1].x*m_Viewport.Width/2 +
                                           m_Viewport.X+m_Viewport.Width/2;

                    Points[PointCount].y = -vecScreenPoints[v1].y*m_Viewport.Height/2 +
                                           m_Viewport.Y + m_Viewport.Height/2;

                    Points[PointCount].z = 0.0f;
                    Points[PointCount].w = 1.0f;
                    Points[PointCount].Diffuse = 0xBF547687;
                    PointCount++;
                    continue; // Skip to next vertex
                }
```

Next we check to see if the vertex is in front or behind the plane. If the vertex is in front of the plane it will be above the water line and should be clipped. If it is behind the plane it is below the water line so we will build a screen space vertex from it and add it to our clipped polygon.

```
                if ( Location1 != 1 )
                {
                    Points[PointCount].x = vecScreenPoints[v1].x*m_Viewport.Width/2 +
                                           m_Viewport.X + m_Viewport.Width  / 2;
                    Points[PointCount].y = -vecScreenPoints[v1].y*m_Viewport.Height/2 +
                                           m_Viewport.Y + m_Viewport.Height / 2;
                    Points[PointCount].z = 0.0f;
                    Points[PointCount].w = 1.0f;
                    Points[PointCount].Diffuse = 0xBF547687;
                    PointCount++;
                } // If it's behind the plane
```

We have now added the current vertex to the polygon. If the next vertex does not cause this edge to span the plane then we are done with this loop iteration.

```
                // If the next vertex is not causing us to span the plane then continue
                if ( Location2 == 0 || Location2 == Location1 ) continue;
```

If the next vertex in the edge is on the opposite side of the plane from the current vertex, the edge will have to be clipped to the plane. This will create a new vertex on the plane. This is exactly the same approach we took with the intersection code in the previous function. The only difference is that we convert the point from projection space to screen space as we copy its X and Y values into the new vertex.

```
                // Calculate the intersection point
                D3DXVECTOR2 Direction, Direction2;
                D3DXVECTOR3 vecIntersection = D3DXVECTOR2( 0.0f, 0.0f );
                Direction        = vecScreenPoints[v2] - vecScreenPoints[v1];
                LineLength       = D3DXVec2Dot(&Direction, &vecPlaneNormal );
                Direction2       = vecPointOnPlane - vecScreenPoints[v1];
                DistFromPlane    = D3DXVec2Dot(&Direction2, &vecPlaneNormal);
                vecIntersection  = vecScreenPoints[v1]+ \
                                   Direction*(DistFromPlane/LineLength);
```

```
                    // This is our new vertex
                    Points[PointCount].x =  vecIntersection.x * m_Viewport.Width  / 2 +
                                            m_Viewport.X + m_Viewport.Width  / 2;
                    Points[PointCount].y = -vecIntersection.y * m_Viewport.Height / 2 +
                                            m_Viewport.Y + m_Viewport.Height / 2;
                    Points[PointCount].z = 0.0f;
                    Points[PointCount].w = 1.0f;
                    Points[PointCount].Diffuse = 0xBF547687;
                    PointCount++;

                } // Next Vertex

            } // End if not coplanar
```

Our new screen space water polygon is now either built at this point or was completely clipped. All that is left to do is render it.

```
            if ( PointCount > 2 )
            {
                // Setup states
                pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_DIFFUSE );

                pD3DDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
                pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_DIFFUSE );
                pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1 );
                pD3DDevice->SetRenderState( D3DRS_SRCBLEND , D3DBLEND_SRCALPHA );
                pD3DDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
                pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );

                // Render polygon
                pD3DDevice->SetFVF( TLITVERTEX_FVF );
                pD3DDevice->DrawPrimitiveUP(D3DPT_TRIANGLEFAN,PointCount-2,
                                            Points, sizeof(CTLitVertex));

                // Reset states
                pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );
                pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
            }

            break;

        } // End Water Case

    } // End Effect Switch

    pD3DDevice->SetRenderState( D3DRS_ZENABLE, D3DZB_TRUE );
}
```

Provided that there are at least three vertices that survived the clipping process, we render the polygon(s). The first thing we do is set the color and alpha operations in stage 0 to sample the color/alpha from the interpolated diffuse color. Then we set the source and destination blend modes so that the alpha weights the blend with the frame buffer.

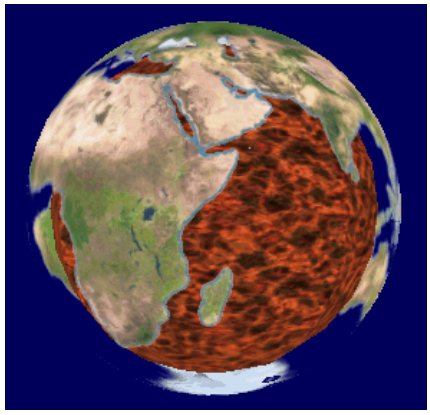
Next we set the flexible vertex flag informing the device that we are using pre-transformed and pre-lit vertices. We will not be requiring the transformation or lighting pipeline here since our vertices are

already in screen space and already colored. We then enabled alpha blending and rendered our array of clipped vertices as a triangle fan. Finally we disabled alpha blending and reset the color operation to sample from the texture again so that everything is put back to the way we found it. At the very end of the function we re-enable the Z-Buffer that we disabled at the start of the function.

We now have a nice alpha blended water effect for our terrain demo and we see how to store and use alpha values in vertex colors. Of course, this demo also taught us a lot more than we might have expected at the outset.

So all of the changes of real significance in this demo took place in the previous functions. Note that under normal circumstances you would probably want to develop a water class that encapsulates all of this functionality rather than take the simplistic approach of having the terrain and camera classes manage the process. This would make for a good study assignment.

Lab Project 7.2: Alpha Channels and Alpha Testing



In our next project we will render two rotating textured spheres; a smaller sphere placed inside a larger sphere. Without some form of alpha processing we would normally not be able to see the smaller sphere because it would be completely occluded by the outer sphere polygons. In this project the inner sphere will have a lava texture mapped to it and the outer sphere will have a texture of the planet Earth mapped to it. The texture map of the Earth used to map to the outer sphere includes an alpha channel. Each pixel in the image has an alpha value between 0 and 255. The sections of the texture representing the oceans have the most transparent alpha values (lower values) while the texels that are part of the land masses have higher alpha values. We will set up the texture stages such that when the outer sphere is rendered, the alpha value for each pixel is sampled from the corresponding texel in the texture map. In this demo we are not using alpha blending. Instead we will use alpha testing to reject pixels with low alpha values. The outer sphere will find that its ocean pixels will be rejected by the test while its land mass pixels pass are rendered. Note that even though the land masses may include pixels with alpha values less than 255, since we are only using alpha testing, pixels are either rendered or not -- no alpha blending occurs. As long as the land mass pixels have alpha values greater than or equal to our reference value, they will be rendered fully opaque. Of course, you could render the land masses partially transparent by assigning them alpha values slightly greater than the alpha reference value whilst enabling alpha blending with the **`D3DBLEND_SRCALPHA`** and **`D3DBLEND_INVSRCALPHA`** blending modes.

Creating an Alpha Channel Image

There are many ways that we can place alpha values in a texture surface and some are more troublesome then others. Probably the most difficult way would be to create the image as a normal texture in a paint package and then load that image into a texture format that has an alpha component such as D3DFMT_A8R8G8B8. We could do this by using the D3DCreateTextureFromFileEx function specifying the filename and the alpha surface we desire. Once the image is loaded we could lock the surface of the texture and step through each pixel inserting the alpha value into the color of each texel. This could be done quite easily if we have an image with a transparent color such as black that we wanted to be totally transparent. Recall that the D3DXCreateTextureFromFileEx function allows us to specify a color key.

```
HRESULT D3DXCreateTextureFromFileEx
(
    LPDIRECT3DDEVICE9 pDevice, LPCTSTR pSrcFile, UINT Width, UINT Height,
    UINT MipLevels, DWORD Usage, D3DFORMAT Format, D3DPOOL Pool,
    DWORD Filter, DWORD MipFilter, D3DCOLOR ColorKey, D3DXIMAGE_INFO *pSrcInfo,
    PALETTEENTRY *pPalette,
    LPDIRECT3DTEXTURE9 *ppTexture
);
```

We can now better understand what the *ColorKey* parameter does. We specify a 32-bit ARGB color and the function will search for a match. If one is found, that pixel alpha component will be set to totally transparent. The alpha component is also significant in the color that we specify so it is important that when you are loading a totally opaque image (an image without an alpha channel) that you set the alpha component of this color to 0xFF. Otherwise color tests will fail to find a matching RGB. As an example, if we wanted full intensity green pixels in an opaque image to become totally transparent, we would specify a color key of 0xFF00FF00.

Swapping a particular color in a texture for a totally transparent color is certainly easy and convenient. But it is obviously very limited in scope. Fortunately, most modern paint packages allow you to create images with an alpha channel and save the resulting image out to a file format that supports alpha (such as .tga or .png). When using a paint package to create an alpha channel image we can literally just paint the alpha information into the image just as we do with the more usually RGB components. The appendices for this lesson include a brief tutorial on creating an alpha channel for a texture image using Jasc's Paint Shop Pro™. Applications such as Adobe Photoshop™ also support this feature, so if that is your preferred editing package then check the accompanying documentation for implementation details.

After we have created a texture file with per-pixel alpha information, we can use the D3DX texture loading functions to load the image data straight into a surface that supports alpha channels and start using it immediately.

There is actually very little new code to examine in this demo since we are essentially just enabling a few render and texture stage states. The CScene class will load the two spheres from an IWF file created using GILES™. GILES™ includes a spherical texture wrapping feature so this is how the

texture coordinates were generated for each of the spheres. The application uses two textures and a single IWF file stored in the application's Data folder.

CGameApp::SetupRenderStates

This function sets up the texture stage states to take the color from the texture and modulate it with the interpolated diffuse vertex color. We also setup the alpha pipeline for texture stage 0 to sample the per-pixel alpha value from the alpha component of the pixel in the texture. Then we setup an alpha testing reference value and the comparison function we wish to use. In this project, any alpha value greater than or equal to 207 will pass the test.

```
void CGameApp::SetupRenderStates()
{
    // Validate Requirements
    if (!m_pD3DDevice || !m_pCamera ) return;

    // Test the device capabilities.
    if (!TestDeviceCaps( )) { PostQuitMessage(0); return; }

    // Setup our D3D Device initial states
    m_pD3DDevice->SetRenderState( D3DRS_ZENABLE, D3DZB_TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_DITHERENABLE,  TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_SHADEMODE, D3DSHADE_GOURAUD );
    m_pD3DDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_NONE );
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, FALSE );

    // Set up sampler states.
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_MINFILTER    , m_MinFilter );
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_MAGFILTER    , m_MagFilter );
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_MIPFILTER    , m_MipFilter );
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_MAXANISOTROPY, m_Anisotropy );

    // Set texture / addressing / color ops
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLOROP  , D3DTOP_MODULATE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP  , D3DTOP_SELECTARG1 );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );

    // Enable alpha testing
    m_pD3DDevice->SetRenderState( D3DRS_ALPHAREF , (DWORD)0x000000CF );
    m_pD3DDevice->SetRenderState( D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL );

    // Set fill mode
    m_pD3DDevice->SetRenderState( D3DRS_FILLMODE, m_FillMode );

    // Setup our vertex FVF code
    m_pD3DDevice->SetFVF( LITVERTEX_FVF );

    // Update our device with our camera details (Required on reset)
    m_pCamera->UpdateRenderView( m_pD3DDevice );
    m_pCamera->UpdateRenderProj( m_pD3DDevice );
```

We set the reference value to 207 instead of 255 because the land mass pixels do not all have fully opaque alpha values. The general range is between 207 and 255. The ocean pixels have much lower alpha values between 0 and 40. Setting the alpha reference value to 207 makes certain that we allow the land mass pixels to pass the test, while masking out the ocean pixels. Notice that we have not enabled alpha testing here. This is because when we render the scene we will only want alpha testing enabled for the outer sphere. The inner sphere does not have an alpha channel in its texture so there is no need to test it. It would always pass the test because an alpha value of 0xFF would be the placeholder value returned from the sampling of the texture in the texture stage. Generally speaking, unnecessary per-pixel operations should be disabled when they are not needed.

Finally, notice the call to TestDeviceCaps() at the top of the function. We learned in Chapter Six that this function can search for the best standard texture format and the best alpha texture format supported by the device. It stores them in the CGameApp member variables m_TextureFormat and m_AlphaFormat. These formats are then passed to the scene using CScene::SetTextureFormat where they are stored. This enables the CScene class to load the textures in the optimal surface formats when it loads the IWF file.

```
     // Inform texture loading objects which format to use
     m_Scene.SetTextureFormat( m_TextureFormat, m_AlphaFormat );

     // Set up the menu item selections (Which may have changed during device validations)
     SelectMenuItems();
}
```

CScene::LoadScene

There is nothing new in this function. In fact it is merely a stripped down version of the LoadScene function that we used in previous lessons. It loads the IWF file using the CFileIWF object and calls ProcessMeshes to extract the polygons and ProcessTextures to load the textures.

```
bool CScene::LoadScene( TCHAR * strFileName )
{
    CFileIWF File;

    // File loading may throw an exception
    try
    {
        // Attempt to load the file
        File.Load( strFileName );

        // Copy over the textures we want from the file
        if (!ProcessTextures( File )) return false;

        // Now process the meshes and extract the required data
        if (!ProcessMeshes( File )) return false;

        // Allow file loader to release any active objects
        File.ClearObjects();
    } // End Try Block
```

```
    // Catch any exceptions
    catch (...)
    {
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

CScene::ProcessTextures

This function loads the textures into alpha supported texture surfaces by specify the **m_fmtAlpha** pixel format to the **`D3DXCreateTextureFromFileEx`** function. Preferably this will be a compressed alpha pixel format that is supported by the current device (determined in the CGameApp::TestDeviceCaps function).

```
bool CScene::ProcessTextures( const CFileIWF& File )
{
    ULONG i;
    char  FileName[MAX_PATH];

    // Allocate enough room for all of our textures
    m_pTextureList = new LPDIRECT3DTEXTURE9[ File.m_vpTextureList.size() ];

    if ( !m_pTextureList ) return false;
    m_nTextureCount = File.m_vpTextureList.size();

    // Loop through and build our textures
    ZeroMemory( m_pTextureList, m_nTextureCount * sizeof(LPDIRECT3DTEXTURE9));
    for ( i = 0; i < File.m_vpTextureList.size(); i++ )
    {
        // Retrieve pointer to file texture
        TEXTURE_REF * pFileTexture = File.m_vpTextureList[i];

        // Skip if this is an internal texture (not supported by this demo)
        if ( pFileTexture->TextureSource != TEXTURE_EXTERNAL ) continue;

        // Build the final texture path
        strcpy( FileName, TexturePath );
        strcat( FileName, pFileTexture->Name );

        // Load the texture from file
        D3DXCreateTextureFromFileEx( m_pD3DDevice, FileName, D3DX_DEFAULT, D3DX_DEFAULT,
                                     D3DX_DEFAULT, 0, m_fmtAlpha, D3DPOOL_MANAGED,
                                     D3DX_DEFAULT, D3DX_DEFAULT, 0,
                                     NULL, NULL, &m_pTextureList[i] );
    } // Next Texture

    // Success!
    return true;
}
```

CScene::Render

When the ProcessMeshes function loads the two sphere meshes from the IWF file, it stores them in a two element array of CMesh objects. Therefore, this render call just has to loop through the two elements in the array and render each mesh. The first mesh in the array is the outer sphere so we enable alpha testing when rendering this mesh and disable it afterwards.

```
void CScene::Render( )
{
    // We render in reverse in our example to ensure that the opaque
    // inner core gets rendererd first.
    for ( long i = 1; i >= 0; i-- )
    {
        if ( i == 0)
        {
             m_pD3DDevice->SetRenderState( D3DRS_ALPHATESTENABLE, TRUE );
        }

        CMesh * pMesh = m_pObject[i].m_pMesh;

        // Set transformation matrix
        m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_pObject[i].m_mtxWorld );

        // Set vertex stream
        m_pD3DDevice->SetStreamSource( 0, pMesh->m_pVertexBuffer, 0, pMesh->m_nStride);

        // Set Properties
        ULONG TextureIndex = pMesh->m_nTextureIndex;
        if ( TextureIndex >= 0 )
        {
            m_pD3DDevice->SetTexture( 0, m_pTextureList[ TextureIndex ] );

        } // End if has texture
        else
        {
            m_pD3DDevice->SetTexture( 0, NULL );
        } // End if has no texture

        // Set indices, and render
        m_pD3DDevice->SetIndices( pMesh->m_pIndexBuffer );
        m_pD3DDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, 0,
                                            pMesh->m_nVertexCount, 0,
                                            pMesh->m_nIndexCount / 3 );

        if ( i == 0)
        {
          m_pD3DDevice->SetRenderState( D3DRS_ALPHATESTENABLE, FALSE );
        }
    } // Next Mesh
}
```

Believe it or not, that is all there is to this project. Try experimenting with the alpha reference value and the alpha comparison function in CGameApp::SetupRenderStates. Make sure that you understand how it all works since this feature is a very important one. Alpha testing allows us to do effects like

chain-link fences, leaves for trees, and a host of other effects that require fully transparent texture regions.

Lab Project 7.3: Alpha Sorting

In Lab Project 7.3 we will load a simple indoor level that includes many partially transparent windows. Our goal will be to implement the sorting and rendering strategies discussed in the text. We will render the alpha polygons in a second pass, sorted back to front. The code changes in this demo are relatively insignificant. As we did in chapters 5 and 6, we will load an IWF file, batch all polygons into light groups, and subsequent texture and material property groups. We will add a new property group to the tree to batch polygons in a light group into alpha and non-alpha groups for easy collection during the first and second passes of our render function.

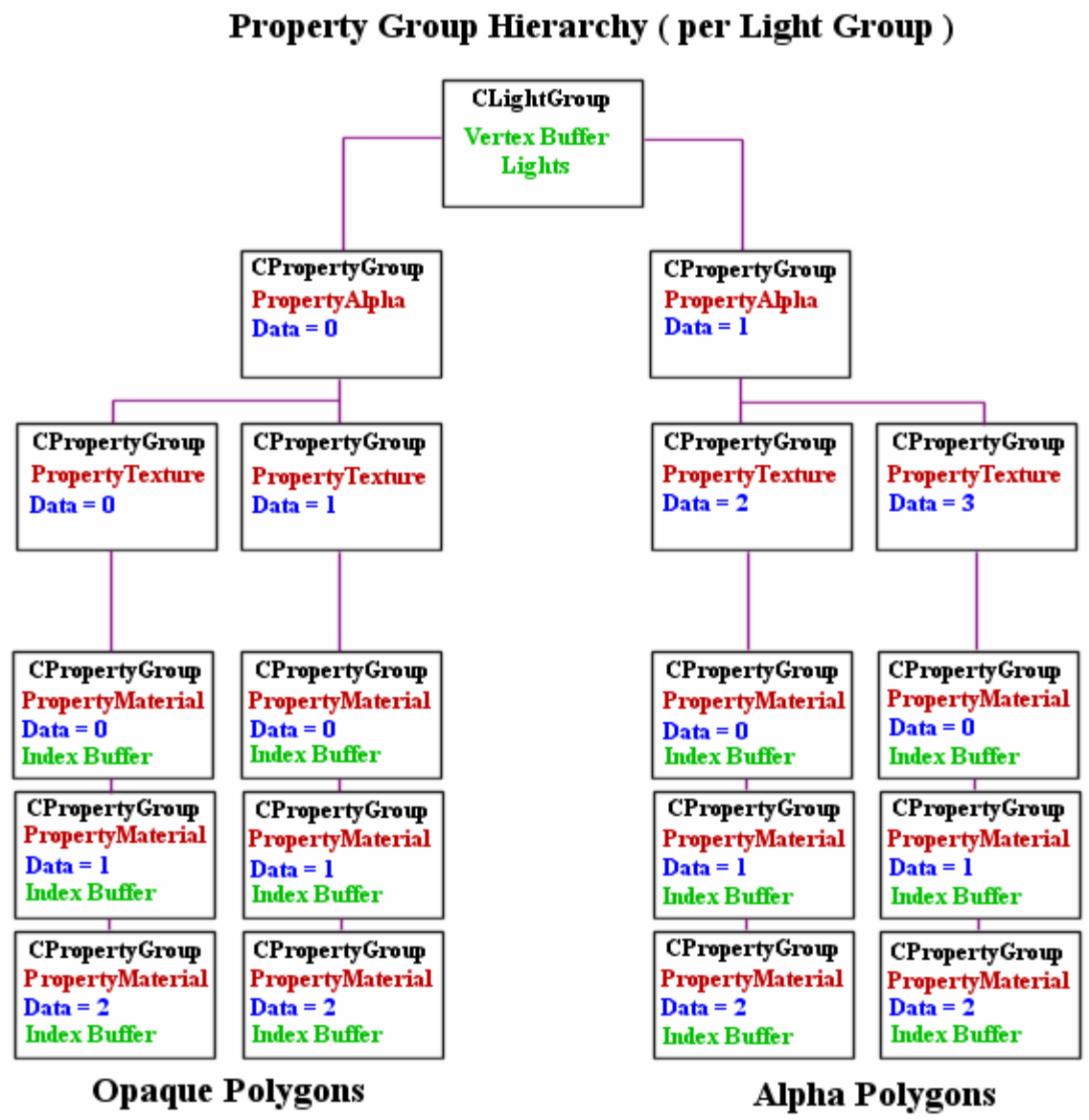
Once our light groups have been successfully compiled, the render loop is fairly simple. We loop through the polygons in our scene and if a polygon is opaque we render it. If it is a transparent polygon we do not render it right away but instead add it to an alpha polygon list. After we have looped through each polygon, we will have all of the opaque polygons rendered into our frame buffer and a list of all alpha polygons waiting to be rendered. If you know for a fact that none of your alpha polygons will ever overlap each other from any point in the level from which the camera can see, you could just loop through this alpha list and render the polygons without regard for ordering. This is because there is no need to worry about the blending order being incorrect since each alpha polygon would only be blended with the frame buffer. However, it is fairly common for alpha polygons to overlap, so we would like to be able to sort them properly before we render them to the frame buffer.

> **Note:** If you are using additive color blending with the frame buffer then sorting is not necessary. In this case A+B+C creates the same color as A+C+B. This is not the case with the common alpha blending mode that we are using (D3DBLEND_SRCALPHA and D3DBLEND_INVSRCALPHA) for the source and destination color blending modes. If for example, you were color blending with the following blending modes for the source and destination:
>
> **Source Blend = D3DBLEND_ONE**
> **Dest Blend = D3DBLEND_ONE**
>
> These states indicate that the color of the polygon we are about to render will be added to the color already in the frame buffer. The rendering order in this case (where we are not scaling based on some arbitrary alpha value) is insignificant. In our final chapter we will write a particle system for effects such as smoke, rain, snow and water. These systems use many hundreds of polygons that often need to be transparent. Fortunately, particle system effects almost always use additive color blending and not alpha blending. Thus we are spared the cost of having to sort hundreds of particles every frame.

Alpha polygons will need to be sorted using the distance from the polygon to the camera as the sort heuristic. Since the camera can move about from frame to frame, we know that this will constantly change the relationship between the camera and the alpha polygons. Thus we cannot sort the polygons as a pre-process; sorting must be done at run-time.

As mentioned, we will now batch our polygons by their alpha state (transparent or not) in addition to the light group, texture, and material batching. We will add a few functions to our CScene class to add alpha polygons to the hash table, and to render the hash table itself after the opaque polygons have been rendered by the main CScene::Render function. Much of this was discussed in the text, so refer back if you do not recall how the alpha sorting hash table works at a high level.

CScene::ProcessMeshes will also undergo a few minor changes. Recall that this is the function that is called by CScene::LoadScene. It extracts each IWFSurface from each IWFMesh and assigns it to a light group based on its light contribution results. This function will now need to test whether the surface is an alpha surface or not and assign it to an alpha property group. This is the first level down from the light group. This means, at most, each light group will have two child property groups with the ID of PROPERTY_ALPHA. This property group type contains no polygons. It stores either a 0 or 1 in its property data member defining the group as an alpha group or an opaque group respectively. Each alpha group has an array of property groups of type PROPERTY_TEXTURE which contain the texture index used by all polygons stored in this group. This property group also stores no polygons directly. Instead it stores an array of child property groups with the PROPERTY_MATERIAL type containing the material used by all polygons stored at this group. This property group contains the index buffer for our triangles. At render time, the triangles in that index buffer will all belong to the same light group, have the same alpha property, and use the same texture and material. Ultimately all we have done is add another node to our property group hierarchy. The hierarchy is shown in the next image:



Virtually all of the changes to the code in this project are contained inside the CScene class. We need to add the hash table as well as a few housekeeping functions to add polygons to the table.

```
const ULONG SORT_HASH_SIZE = 1000;   // Size of the alpha sorting hash table

class CScene
{
       …
       …
      ASORT_ITEM   *m_pSortContainer[SORT_HASH_SIZE];  // Hash table for alpha sorting
```

SORT_HASH_SIZE defines our hash table size for this demo (1000). Each element in the hash array is a pointer to an ASORT_ITEM structure. The ASORT_ITEM structure contains all of the information for a single alpha polygon. It will be filled out for each alpha polygon during the CScene::Render call. Once we find an alpha polygon in the render loop, we will allocate an ASORT_ITEM structure and fill in the polygon information such as its squared distance from the camera, the light group it belongs to, the index in the light group vertex buffer where the polygon vertices begin in the index buffer, the number of vertices in the polygon, the texture index and the material index, and a pointer to the index buffer for easy access. The structure also stores a next pointer so that each element in the hash table can exist as a linked list of ASORT_ITEM structures. They will be connected to any other ASORT_ITEMS in the linked list ordered by distance. This is necessary to resolve collisions when the polygon hash keys map to the same index. The ASORT_ITEM structure is shown below.

```
typedef struct _ASORT_ITEM               // Alpha sorting item
{
    LPDIRECT3DINDEXBUFFER9  Indices;     // Index buffer pointer
    USHORT      IndexStart;              // The starting index for this primitive
    USHORT      BaseVertex;              // BaseVertexIndex passed to DP calls
    USHORT      VertexStart;             // The starting vertex where this primitive exists
    USHORT      VertexCount;             // Number of verts in the above batch
    long        TextureIndex;            // Texture to be applied to this primitive
    long        MaterialIndex;           // Material to be applied to this primitive
    float       Distance;                // Distance from the camera
    CLightGroup  *LightGroup;            // Light group to which this primitive belongs
    _ASORT_ITEM  *Next;                  // Next item in linked List
} ASORT_ITEM;
```

Our housekeeping functionality will be:

```
      void    AddAlphaSortItem     ( ASORT_ITEM * pItem, ULONG HashIndex );
      void    RenderSortedAlpha    ( );
      ….
      ….
};
```

The AddAlphaSortItem will be called from the main render function whenever an alpha polygon is encountered. The render function will pass the polygon information in the ASORT_ITEM and along with the hash table index where the data should be added. This function is responsible for adding the ASORT_ITEM to the correct index in the hash table. If there is already a linked list of structures stored there, it will locate the correct position in the list so that back to front ordering is maintained.

RenderSortedAlpha is called after all opaque polygons have been rendered and the hash table has been filled. This function loops through the table starting from the bottom of the array and working towards the top, rendering as it goes. When it returns, the alpha polygons will have been rendered in back-to-front order and will have been correctly blended in the frame buffer. The main render function now has to detect alpha polygons, calculate the squared distance, calculate the hash table index, and finally copy the polygon information into an ASORT_ITEM structure.

In order to calculate the distance from the camera to the polygon during the render pass, we use the pre-calculated center points of the alpha polygons. We will calculate these for each alpha face as they are added to the index buffer in their material property group to which they belong. As the above diagram shows, only the material property groups (the leaf nodes of our batch tree) contain index buffers. Thus we will use the material groups to store our center points. Note that these will only be stored in material groups that are descendants of an alpha node group. We have no need for the center points of opaque polygons in this particular demo.

Each polygon in the property group index buffer will have a corresponding center point stored as a vector. So our property group will now contain an additional pointer of type D3DXVECTOR3 that will be used to allocate and point to an array of center points, one for each face. Below we see the new member variable in the CPropertyGroup class. We also added a new member to the PROPERTY_TYPE enumerated type called PROPERTY_ALPHA to identify alpha groups.

```
class CPropertyGroup
{
   enum PROPERTY_TYPE
   { PROPERTY_NONE = 0, PROPERTY_MATERIAL = 1, PROPERTY_TEXTURE = 2, PROPERTY_ALPHA = 3 };
    ….
    D3DXVECTOR3     *m_pCenterPoints;
    ….
    ….
};
```

CScene::ProcessMeshes

```
bool CScene::ProcessMeshes( CFileIWF & pFile )
{
  long i, j, k, l, m, n, TextureIndex, MaterialIndex;
  CLightGroup    * pLightGroup    = NULL;
  CPropertyGroup * pAlphaProperty = NULL;
  CPropertyGroup * pTexProperty   = NULL;
  CPropertyGroup * pMatProperty   = NULL;
```

The first thing we do in this function is call BuildLightGroups to create all light groups and determine which groups polygons are assigned to. When this function returns, each IWFSurface will have the index of the light group to which it belongs temporarily stored in its CustomData member.

```
    // Allocate the light groups, and assign the surfaces to them
    if (!BuildLightGroups( pFile )) return false;
```

We begin adding the polygons to property groups in a hierarchical fashion. The first loop cycles between 1 and 0. When the loop index n equals 1 we will add the alpha polygons to the appropriate property groups. When n is 0 we add the opaque polygons to their appropriate property groups. The reason we counted backwards in this loop is so that alpha property groups are created for the light groups first. This is not a requirement by any means, but it better helps us catch the case where

incorrect results occur when we do not enable alpha blending and the alpha polygons will still need to be rendered first.

```
for ( n = 1; n >= 0; n-- )
{
```

We will now loop in the following type order: texture, material, mesh, and finally, face. Each iteration of these inner loops will extract only polygons that match all current properties and add them to the appropriate group. This is the same batching strategy we used in the previous chapters.

```
// For each texture (including no texture, -1)
for ( l = -1; l < (signed)m_nTextureCount; l++ )
{
  // For each material (including no material, -1)
  for ( m = -1; m < (signed)m_nMaterialCount; m++ )
  {
    // Loop through each surface of each mesh
    for ( i = 0; i < pFile.m_vpMeshList.size(); i++ )
    {
       iwfMesh * pMesh = pFile.m_vpMeshList[i];

       for ( j = 0; j < pMesh->SurfaceCount; j++ )
       {
          iwfSurface * pSurface = pMesh->Surfaces[j];
```

Once we have a pointer to the surface we are currently processing, we will test it against the current properties. First, we see if the surface has the invisible IWF surface flag set. If it does then it will not rendered and we skip it. It never gets added to any of our light groups and is never used by our scene.

```
          if ( pSurface->Style & SURFACE_INVISIBLE ) continue;
```

Our next test looks at the alpha properties of the surface. When a surface in an IWF file has alpha properties, it will store a source blend mode and a destination blend mode describing the alpha blending equation that the level designer intends the engine to use when rendering this surface. Although we are using the standard alpha blending equation in this demo, we can still use the surface blend modes to determine if the face needs alpha processing. If the IWFSurface::Components member has the SCOMPONENT_BLENDMODES flag set then it means that this surface has a Blend Modes array and as such, is an alpha polygon. If not, then it is opaque. We only read the blend modes from the first channel of the surface in this demo. Each source and destination blend mode combination is stored inside a BLEND_MODE structure defined in the header file libIWF. It is a simple two byte structure where the first byte contains the source blend mode number and the second byte describes the destination blend mode number. The IWF specification documentation contains a table of how the values map to DirectX blend modes. We create a local BLEND_MODE variable and use it to read in the blend modes of the first channel of the surface.

```
          // Determine the blend modes we are using
          BLEND_MODE BlendMode = { 0, 0 };
          if((pSurface->Components & SCOMPONENT_BLENDMODES) &&
              pSurface->ChannelCount > 0)
```

```
         BlendMode = pSurface->BlendModes[0];
```

If n=0 then we are currently searching for non-alpha polygons. We will skip the current polygon if it has non zero blend modes in this case because it is intended to be alpha blended. If n=1 and we are processing alpha polygons, we skip any polygons that have zero source and destination blend modes as shown below.

```
        // Skip if it is not in order
        if ( (n == 0 && (BlendMode.DestBlendMode != 0 ||
                         BlendMode.SrcBlendMode !=0 )) ||
              n == 1 && (BlendMode.DestBlendMode == 0 &&
                         BlendMode.SrcBlendMode ==0 ))) continue;
```

If we get this far then we have a polygon that matches the n requirement of our outer loop (alpha vs. opaque). From this point forward we extract the texture and material indices and proceed building the tree as before.

```
        // Determine the indices we are using
        MaterialIndex = -1;
        TextureIndex  = -1;

        if((pSurface->Components & SCOMPONENT_MATERIALS)&&pSurface->ChannelCount > 0)
              MaterialIndex = pSurface->MaterialIndices[0];

        if ((pSurface->Components & SCOMPONENT_TEXTURES)&&pSurface->ChannelCount > 0)
              TextureIndex  = pSurface->TextureIndices[0];

        // Skip if this is not in order
        if ( TextureIndex != l || MaterialIndex != m ) continue;
```

At this point in the code we have a polygon such that ‘n’ describes its alpha state, ‘l’ describes its texture and ‘m’ describes its material. It is now time to add it to the light group to which it belongs. Recall that during the call to BuildLightGroups, each IWF surface has a CLightGroup pointer stored in its CustomData member.

```
        // Retrieve the lightgroup pointer for this surface
        pLightGroup = (CLightGroup*)pSurface->CustomData;
```

Now that we have the light group to which the polygon should belong, we traverse the child property groups. There will at most be only two (alpha and non-alpha). Each immediate child property group will be of the type PROPERTY_ALPHA. As you might expect, our next task is to loop through these property groups until the correct match is found for the current surface. If we do not find a child property group that deals with polygons that match the alpha state of ‘n’ then we need to create a new one.

```
        // Determine if we already have a property group for this alpha state
        // (enabled / disabled only)
        for ( k = 0; k < pLightGroup->m_nPropertyGroupCount; k++ ){
             if ( (long)pLightGroup->m_pPropertyGroup[k]->m_nPropertyData == n ) break;
        }
```

```
        // If we didn't have this property group, add it
        if ( k == pLightGroup->m_nPropertyGroupCount )
        {
             if ( pLightGroup->AddPropertyGroup( ) < 0 ) return false;

             // Set up property group data for primary key
             pAlphaProperty = pLightGroup->m_pPropertyGroup[ k ];
             pAlphaProperty->m_PropertyType  = CPropertyGroup::PROPERTY_ALPHA;
             pAlphaProperty->m_nPropertyData = (ULONG)n;
        }

        // Process for secondary key (texture)
        pAlphaProperty = pLightGroup->m_pPropertyGroup[ k ];
```

The remaining steps are just a similar traversal through the child property groups to find the correct match. We start with textures first:

```
        // Determine if we already have a property group for this texture
        for ( k = 0; k < pAlphaProperty->m_nPropertyGroupCount; k++ )
        {
             if((long)pAlphaProperty->m_pPropertyGroup[k]->m_nPropertyData== TextureIndex)
                  break;
        }

        // If we didn't have this property group, add it
        if ( k == pAlphaProperty->m_nPropertyGroupCount )
        {
            if ( pAlphaProperty->AddPropertyGroup( ) < 0 ) return false;

            // Set up property group data for primary key
            pTexProperty = pAlphaProperty->m_pPropertyGroup[ k ];
            pTexProperty->m_PropertyType  = CPropertyGroup::PROPERTY_TEXTURE;
            pTexProperty->m_nPropertyData = (ULONG)TextureIndex;
        }

        // Process for secondary key (material)
        pTexProperty = pAlphaProperty->m_pPropertyGroup[ k ];
```

Testing for a matching material group is next:

```
        // Determine if we already have a property group for this material
        for ( k = 0; k < pTexProperty->m_nPropertyGroupCount; k++ )
        {
            if((long)pTexProperty->m_pPropertyGroup[k]->m_nPropertyData == MaterialIndex)
                break;
        }

        // If we didn't have this property group, add it
        if ( k == pTexProperty->m_nPropertyGroupCount )
        {
            if ( pTexProperty->AddPropertyGroup( ) < 0 ) return false;

            // Set up property group data for primary key
            pMatProperty = pTexProperty->m_pPropertyGroup[ k ];
            pMatProperty->m_PropertyType  = CPropertyGroup::PROPERTY_MATERIAL;
            pMatProperty->m_nPropertyData = (ULONG)MaterialIndex;
```

```
            pMatProperty->m_nVertexStart  = pLightGroup->m_nVertexCount;
            pMatProperty->m_nVertexCount  = 0;
        }

        // Collect the material property group
        pMatProperty = pTexProperty->m_pPropertyGroup[ k ];
```

Next we call the ProcessIndices function which is also unchanged from previous chapters. This function calculates the indices for the polygons and adds them to the index buffer.

```
        // Process the indices
        if (!ProcessIndices( pLightGroup, pMatProperty, pSurface ) ) return false;
```

If this polygon has the SURFACE_TWO_SIDED style flag set then it means that this polygon should be rendered from both sides. Rather than disable back face culling when rendering such polygons, the ProcessIndices function allows us to pass a boolean parameter indicating that we are adding two sided polygons. In this case, we call the function again and back face polygons will have their indices added again to the index buffer in counterclockwise order. So instead of having one two-sided polygon, we store two one-sided polygons in our index buffers with opposite winding orders. These will now render correctly without the need to adjust the culling render state.

```
            if ( pSurface->Style & SURFACE_TWO_SIDED )
            {
                // Two sided surfaces have back faces added manually
                if (!ProcessIndices( pLightGroup, pMatProperty, pSurface, true ) )
                      return false;
            }

            // Process vertices
            if (!ProcessVertices( pLightGroup, pMatProperty, pSurface ) ) return false;

          } // Next Surface
        } // Next Mesh
      } // Next Material
    } // Next Texture
} // Next alpha type
```

The ProcessVertices function is finally called at the bottom of the inner loop to add the vertices of the surface to the light group vertex buffer.

By the time we exit the alpha loop, all scene polygons will exist in their correct property groups ready for rendering. Before we return, we loop through each IWFSurface in our scene and reset the CustomData member back to zero. We temporarily used it to store light groups, but when the destructor is called it will try to de-allocate the CustomData if it is not set to null. This would free our light groups from memory.

```
    // Clear the custom data pointer so that it isn't released
    for ( i = 0; i < pFile.m_vpMeshList.size(); i++ ){
        iwfMesh * pMesh = pFile.m_vpMeshList[i];
        for ( j = 0; j < pMesh->SurfaceCount; j++ ) pMesh->Surfaces[j]->CustomData = NULL;
    }
```

```
    // Success!!
    return true;
}
```

When program flow returns to the CScene::LoadScene function, it will loop through each of the light groups that were created and call BuildBuffers. This call creates the vertex buffers and index buffers that are needed for rendering.

CPropertyGroup::BuildBuffers

The CPropertyGroup::BuildBuffers function has been slightly modified for this demo. We added some new code to allocate the center points array and calculate the center point of each polygon in the index buffer.

The first thing the function does is determine if this is an alpha property group. If so it checks the m_nPropertyData member to discover whether the alpha group holds alpha or non-alpha polygons. If the m_nPropertyData member is not zero then the alpha polygons will need to be rendered in a sorted fashion. The Sortable parameter will be passed down through the child BuildBuffer calls so that eventually, any material property groups (which contain the index buffers) that are children of an alpha property group that contains alpha polygons, will receive this flag informing the function that a center point array will need to compiled.

```
bool CPropertyGroup::BuildBuffers(LPDIRECT3DDEVICE9 pD3DDevice, bool HardwareTnL,
                                  boolReleaseOriginals, CLightGroup * pLightGroup /* = NULL */,
                                  bool Sortable /* = false */ )
{
    HRESULT     hRet    = S_OK;
    USHORT     *pIndex  = NULL, i;
    ULONG       ulUsage = D3DUSAGE_WRITEONLY;

    // Is this an alpha enable property, override the sortable flag
    if ( m_PropertyType == PROPERTY_ALPHA && m_nPropertyData != 0 ) Sortable = true;

    // Keep original indices if sortable
    if ( Sortable ) ReleaseOriginals = false;
```

If the property group has a non-zero index count then it means it is a material property group at the bottom of our hierarchy containing the index buffer. If this is the case then any current index buffer will need to be released.

```
    // Allocate center point array and build, if we store indices here
    if ( m_nIndexCount > 0 )
    {
        // Release any previously allocated vertex / index buffers
        if ( m_pIndexBuffer ) m_pIndexBuffer->Release();
        m_pIndexBuffer = NULL;
```

If the Sortable flag is set then this material property group has a parent alpha property group that contains alpha polygons. As such, the center point array of this property group will need to be allocated to hold a center point for each triangle in the index buffer. As we are rendering indexed triangle lists, simply dividing the index count of this property group by 3 will provide us with the total number of triangles in the index buffer. This will be the number of 3D vectors we will need to allocate. This is done only if it is a Sortable buffer because material groups which do not contain alpha polygons do not need to store center points.

```
// Is this a sortable buffer ?
if ( Sortable )
{
   // Bail if there is no light group, or vertex data
   if ( !pLightGroup || !pLightGroup->m_pVertex )    return false;

   if ( m_pCenterPoints ) delete []m_pCenterPoints;
   m_pCenterPoints = new D3DXVECTOR3[ m_nIndexCount / 3 ];

   if (!m_pCenterPoints) return false;

   // Build the center point data
   D3DXVECTOR3 CenterPoint;
   for ( i = 0 ; i < m_nIndexCount; i += 3 )
   {
       CenterPoint  = (D3DXVECTOR3&)pLightGroup->m_pVertex[m_pIndex[i]+m_nVertexStart];
       CenterPoint += (D3DXVECTOR3&)pLightGroup->m_pVertex[m_pIndex[i+1]+m_nVertexStart];
       CenterPoint += (D3DXVECTOR3&)pLightGroup->m_pVertex[m_pIndex[i+2]+m_nVertexStart];
       m_pCenterPoints[ i / 3 ] = CenterPoint / 3.0f;
   }
} // End if sortable
```

We calculated triangle center pointse by adding together the vertex positions and dividing by 3. Next we create the index buffer, lock it, and copy the indices of this property group into it. We release the original indices (non index buffer indices) created during the LightGroup/PropertyGroup building process.

```
    // Should we use software vertex processing ?
    if ( !HardwareTnL ) ulUsage |= D3DUSAGE_SOFTWAREPROCESSING;

    // Create our index buffer
    pD3DDevice->CreateIndexBuffer( sizeof(USHORT) * m_nIndexCount, ulUsage,
                                   D3DFMT_INDEX16, D3DPOOL_MANAGED,
                                   &m_pIndexBuffer, NULL );

    // Lock the index buffer ready to fill data
    hRet = m_pIndexBuffer->Lock(0, sizeof(USHORT)*m_nIndexCount, (void**)&pIndex, 0);
    if ( FAILED( hRet ) ) return false;

    // Copy over the index data
    memcpy( pIndex, m_pIndex, sizeof(USHORT) * m_nIndexCount );

    // We are finished with the index buffer
    m_pIndexBuffer->Unlock();
```

```
    // Release old data if requested
    if ( ReleaseOriginals )
    {
        // Release our components
        if ( m_pIndex ) delete []m_pIndex;
        m_pIndex = NULL;

    } // End if ReleaseOriginals
} // End if indices
```

The code above is only executed when the property group has indices stored -- which is only true if we are at the bottom of the hierarchy in a material property group. The material property groups will not have any child property groups but the alpha and texture groups will. Therefore, at the end of the function we loop through each child of the property groups and call the BuildBuffers function to propagate the buffer building process down through the hierarchy. This is also how we pass the Sortable flag (determined at the alpha property group level) through the hierarchy so that material property groups know whether or not they need to create a center points array.

```
    // Build buffers for each child property group
    for ( i = 0; i < m_nPropertyGroupCount; i++ )
    {
        if(!m_pPropertyGroup[i]->BuildBuffers(pD3DDevice,HardwareTnL,
                                              ReleaseOriginals,
                                              pLightGroup, Sortable ))
            return false;
    }

    // Success
    return true;
}
```

CGameApp::SetupRenderStates

Before we move on to rendering, take a moment to examine CGameApp::SetupRenderStates. This is where the alpha blending states are set up and the texture stages are configured. This information will not change in the main rendering loop.

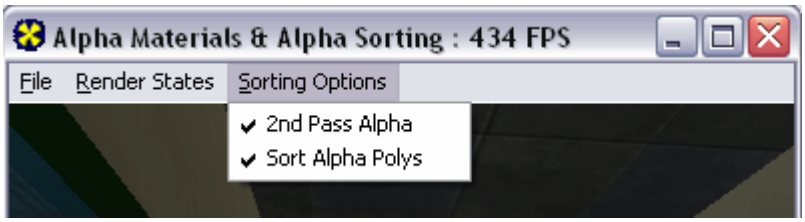
```
    // Set texture / addressing / color ops
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLOROP  , D3DTOP_MODULATE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 0 );

    // Set Alpha Ops
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG2, D3DTA_DIFFUSE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP  , D3DTOP_MODULATE );

    // Select alpha blending states
    m_pD3DDevice->SetRenderState( D3DRS_SRCBLEND , D3DBLEND_SRCALPHA    );
    m_pD3DDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
```

We instruct the device to take the colors from the vertex and the sampled texel from the texture bound to stage 0 and modulate them to create the final color. The alpha states basically do the same thing. Usually, alpha values will be stored in either the vertex/material or the texture. However, it is possible for a polygon to have alpha values stored at the vertices and have a texture mapped to it that also has an alpha channel. Our texture states inform the device that if there is alpha in the vertex and the texture, to modulate the values to create one combined alpha value used for frame buffer blending. If alpha is stored in only one of the two alpha sources, then the default value for the missing alpha source will be assigned a default value of 1.0 (opaque) such that if A is the alpha source, A*1 = A.

Finally, we set the common blend modes as described earlier in this lesson so that the alpha value is used to weight the blending between alpha polygons and the frame buffer. These blend modes will not actually take effect until we enable alpha blending in the CScene::Render function after we have rendered the opaque polygons and are about to render the alpha polygons from the hash table.

CScene::Render

This function draws the scene. For each light group we loop through each of the child property groups and record whether the group contains alpha or opaque polygons. Then we loop through each of the texture property groups and get the texture index, then through each of the material property groups of the texture group where we finally get access to the material. At this point, we either set the texture and the material and render the polygon if it opaque, or calculate the squared distance between the camera position and the polygon center point and use this distance to generate a hash table index if it is transparent. We then add the transparent polygon to the hash table. Once done, all opaque polygons will be rendered and the final pass will render the alpha polygons in the hash table in back-to-front order with alpha blending enabled.



The application has two menu items that allow us to disable the alpha pass. In that case the alpha polygons will not be rendered after the opaque polygons but will be rendered in the order they are stored inside the light group tree. This allows us to see what happens when alpha polygons are rendered in no particular sorted order. With 2nd Pass Alpha enabled, the opaque polygons are rendered first and alpha polygons rendered afterwards. Here we have a choice of whether to sort the alpha polygons back to front or to render the alpha polygons in no particular order in the second pass. These options slightly complicate the render function as we need to handle the alpha polygons in different ways depending on the menu items selected, but it is a good exercise to have this feature so that we can see that only with 2nd pass alpha enabled with polygon sorting will the alpha polygons truly render correctly.

In our previous lessons we discussed the overall operation of light group rendering. We first loop through each light group in the outer loop and disable any lights that are set beyond the number of lights in the current light group. That is, if the previous light group used 10 lights and this light group only uses 5, we will disable light slots 5 through 9. We don't have to disable light slots 0 through 4 because these will be replaced by the 5 lights in the current light group as shown below.

```
void CScene::Render( CCamera & Camera )
{
    ULONG         i, j, k, l, m;
    CLightGroup * pLightGroup = NULL;
    ULONG       * pLightList  = NULL;

    // Loop through each light group
    for ( i = 0; i < m_nLightGroupCount; i++ )
    {
        // Set active lights
        pLightGroup = m_ppLightGroupList[i];
        pLightList  = pLightGroup->m_pLightList;
        for ( j = m_nReservedLights; j < m_nLightLimit; j++ )
        {
            if ( (j - m_nReservedLights) >= (pLightGroup->m_nLightCount ) )
            {
               m_pD3DDevice->LightEnable( j, FALSE );
            }
            else
            {
               // Set this light as active
               m_pD3DDevice->SetLight(j, &m_pLightList[pLightList[j - m_nReservedLights]]);
               m_pD3DDevice->LightEnable( j, TRUE );
            }
        } // Next Light
```

The device now has the current lights set. Remember that the m_nReserved member variable describes how many light slots our application wanted to reserve for use by dynamic lights. If we have reserved two lights, light slots 0 and 1 will not be used by the light group system. Next, we bind the current light group vertex buffer to stream zero so its vertices are ready for rendering.

```
     // Set vertex stream
     m_pD3DDevice->SetStreamSource(0,pLightGroup->m_pVertexBuffer, 0,sizeof(CVertex));
```

The light group will have either one or two alpha groups depending on whether this light group contains both alpha and opaque polygons at the lowest level of its tree (the material property group level). Therefore we need to loop through each direct child property group of this light group. These will be property groups of type PROPERTY_ALPHA which will have an m_nPropertyData member set to either 0 or 1 describing this group as containing either opaque or alpha polygons respectively.

```
     // Now loop through and render the associated property groups
    for ( j = 0; j < pLightGroup->m_nPropertyGroupCount; ++j )
    {
        CPropertyGroup * pAlphaProperty = pLightGroup->m_pPropertyGroup[j];
        ULONG AlphaEnabled = pAlphaProperty->m_nPropertyData;
```

In the code above we get a pointer to the current alpha property group we are traversing and store whether or not this is a property group that contains alpha polygons (=1) or opaque polygons (=0) in the AlphaEnabled variable.

Next we need to loop through each child of the alpha property group. These will be texture property groups that store texture indices used by all polygons in the group. In the following code, we get a pointer to the current texture property group and store the texture index in the local TextureIndex variable.

```
    // Render child property group
    for ( l = 0; l < pAlphaProperty->m_nPropertyGroupCount; ++l )
    {
        CPropertyGroup * pTexProperty = pAlphaProperty->m_pPropertyGroup[l];
        long TextureIndex = (long)pTexProperty->m_nPropertyData;
```

If the AlphaEnabled local variable is set to zero then it means the alpha property group we are current rendering contains opaque polygons. These will be rendered immediately. In that case, we can set the texture used by the property group in stage 0. If AlphaEnabled equals 1 then we are rendering a group filled with alpha polygons. In this case we will render it later and will not set its texture at this point. Instead we set the texture in stage 0 to NULL. Notice that we also set the texture if the CGameApp::m_bSecondPassAlpha variable is set to false. This means that the user has decided (via the menu options discussed previously) that they want the alpha polygons rendered in the first pass without any consideration for rendering order. If this is the case, the alpha polygons are rendered at the same time as opaque polygons and we will set the property groups texture.

```
            // Alpha polys are simply collected
            if ( AlphaEnabled == 0 || GetGameApp()->m_bSecondPassAlpha == false )
            {
                // Set Properties
                if ( TextureIndex >= 0 )
                {
                    m_pD3DDevice->SetTexture( 0, m_pTextureList[ TextureIndex ] );
                }
                else
                {
                    m_pD3DDevice->SetTexture( 0, NULL );
                }
            } // End if alpha primitives
```

Each texture property group will contain an array of one or more child material property groups. These contain the material index used by all of the polygons stored there. So we need to loop through each material property group and if we are rendering an alpha group that contains opaque polygons, render them immediately. Again, we also render the polygons immediately if m_bSecondPassAlpha has been set to false (usually with incorrect blending results). If we are rendering alpha polygons immediately (m_bSecondPassAlpha = false) then we must still remember to enable alpha blending before we render. Also, while not strictly necessary in this demo, we disable Z buffer writing when rendering alpha polygons so that the alpha polygons will not occlude anything in the depth buffer. We reset both of these states after the polygons have been rendered.

```
// Render child property group
for ( k = 0; k < pTexProperty->m_nPropertyGroupCount; ++k )
{
       CPropertyGroup * pMatProperty = pTexProperty->m_pPropertyGroup[k];

       if ( AlphaEnabled == 0 || GetGameApp()->m_bSecondPassAlpha == false )
       {
           // Enable alpha blending if we are not performing 2nd pass alpha
           if ( AlphaEnabled )
           {
               m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
               m_pD3DDevice->SetRenderState( D3DRS_ZWRITEENABLE, FALSE );
           }

           // Simply render opaque polygons
           m_pD3DDevice->SetMaterial(&m_pMaterialList[(long)pMatProperty->m_nPropertyData]);
           m_pD3DDevice->SetIndices( pMatProperty->m_pIndexBuffer );
           m_pD3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST,
                                              pMatProperty->m_nVertexStart,
                                              0, pMatProperty->m_nVertexCount, 0,
                                              pMatProperty->m_nIndexCount / 3 );

           // Enable alpha blending if we are not performing 2nd pass alpha
           if ( AlphaEnabled )
           {
              m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );
              m_pD3DDevice->SetRenderState( D3DRS_ZWRITEENABLE, TRUE );
           }
       } // End if opaque primitives
```

If the current alpha group contains alpha polygons and we have enabled 2nd Pass Alpha in the menu, then the polygons belonging to this material group will not be rendered and will be added to the hash table instead. If CGameApp::m_bSortedAlpha is set to true (the default state) then the hash table index will be generated based on distance to the polygon center point from the camera. Polygons are inserted into the hash table in an ordered way based on this distance. If m_bSortedAlpha has been set to false then all alpha polygons will simply be added to the linked list stored at hash table index 0.

```
       else
       {
           D3DXVECTOR3 vecCameraPos = Camera.GetPosition();
           float       fMaxDistance = powf( Camera.GetFarClip(), 2 );
           float       fDistance    = 0.0f;
           long        Index        = 0;
```

We are going to calculate the squared distance from the camera to each triangle center point and map it to an index using the squared far plane distance. We retrieve the camera's far plane distance and raise it to a power of 2 (to square it) and store the result in the fMaxDistance local variable.

Next, we need to loop through each triangle in the index buffer of the material property group and calculate its hash table index. This only happens if the user has enabled the sorting of alpha polygons (the default state).

```
// Alpha primitives must be collected for sorting and subsequent rendering.
for ( m = 0; m < pMatProperty->m_nIndexCount; m += 3 )
{
       // Are we sorting them ?
       if ( GetGameApp()->m_bSortAlphaPolys )
       {
          // Calculate the distance to the center point (no need to sqrt)
          fDistance=D3DXVec3LengthSq(&(pMatProperty->m_pCenterPoints[(m/3)]-vecCameraPos));

          // Transform this into an index within the range supported by our hash table
          Index = (long)((fDistance / fMaxDistance) * (SORT_HASH_SIZE - 1));

          // Bail if this is out of range
          if ( Index < 0 || Index >= SORT_HASH_SIZE ) continue;

       } // End if sort alpha polys
```

For the hash table index calculation above we start by subtracting the camera position from the center point of the current face we are processing. We calculate the squared length of the resulting vector, which gives us the squared distance from the camera position to the center point of the current triangle. Remember that every three indices in the index buffer is a triangle. Therefore, dividing the current loop counter m by three gives us the index of the triangle in the index buffer we are processing. We use this index to retrieve the pre-calculated center point for that triangle from the material property group center points array. Once we have the squared distance from the camera to the triangle center point we divide the squared distance by the squared far plane distance to scale the distance into the 0.0 to 1.0 range. A center point at the very far side of the scene (touching the far clip plane) will generate a floating point index of 1.0. Next we multiply the floating point index by the size of our hash table minus 1. Thus if the hash table has 1000 elements, the floating point index would be mapped from the 0.0 to 1.0 range to the 0 to 999 range. This final integer is the actual index used to assign the triangle to the hash table (the hash key). After we have calculated the final index we check to see if it is larger than the capacity of the hash table array. If it is, then it means that this polygon must be beyond the far plane and will not be rendered by the pipeline anyway. Therefore we can skip this polygon and move on to the next iteration of the loop to process the next triangle in the index buffer (if one exists).

Is alpha sorting is enabled we will have correctly calculated the index value we will use to add the triangle to the hash table. If alpha sorting is not enabled, the index value will equal 0 and the alpha polygons will all be added to a linked list at hash table element 0. Furthermore, because the fDistance variable will also be 0, polygons will not be sorted in the linked list since they will all have the same distance values. This will allow us to see what the alpha polygons look like if they are rendered in a second pass, but not rendered in back to front order.

To add the triangle to the hash table we allocate a new ASORT_ITEM structure and fill in all the members with the triangle details:

```
       // Allocate a new sort item for the container
       ASORT_ITEM * pSortItem = new ASORT_ITEM;

       if (!pSortItem) return;
       ZeroMemory( pSortItem, sizeof(ASORT_ITEM));
```

```
        // Fill out its values
        pSortItem->LightGroup    = pLightGroup;
        pSortItem->Indices       = pMatProperty->m_pIndexBuffer;
        pSortItem->TextureIndex  = TextureIndex;
        pSortItem->MaterialIndex = (long)pMatProperty->m_nPropertyData;
        pSorttItem->Distance     = fDistance;
```

We also store minimum and maximum vertices used by this triangle in the ASORT_ITEM structure so that we can use them when rendering. This enables us to index into the light group vertex buffer correctly. Recall that the minimum vertex index and the vertex count are used by a software device to transform a block of vertices in one go. Once we have filled out the information for this triangle, we call the CScene::AddSortItem function. The first parameter is a pointer to an ASORT_ITEM structure that will be added to the hash table, and the second parameter is the hash table index for this triangle.

```
        // Loop through the three indices for this tri and find the
        // minimum and maximum vertex indices.
        USHORT MinIndex = 0xFFFF, MaxIndex = 0, IndexVal;

        IndexVal = pMatProperty->m_pIndex[m];
        if ( IndexVal < MinIndex ) MinIndex = IndexVal;
        if ( IndexVal > MaxIndex ) MaxIndex = IndexVal;

        IndexVal = pMatProperty->m_pIndex[m + 1];
        if ( IndexVal < MinIndex ) MinIndex = IndexVal;
        if ( IndexVal > MaxIndex ) MaxIndex = IndexVal;

        IndexVal = pMatProperty->m_pIndex[m + 2];
        if ( IndexVal < MinIndex ) MinIndex = IndexVal;
        if ( IndexVal > MaxIndex ) MaxIndex = IndexVal;

        // Store these properties to pass to DrawIndexedPrimitive
        pSortItem->IndexStart    = m;
        pSortItem->BaseVertex    = pMatProperty->m_nVertexStart;
        pSortItem->VertexStart   = MinIndex;
        pSortItem->VertexCount   = (MaxIndex - MinIndex) + 1;

        // Add this item to the hash table
        AddAlphaSortItem( pSortItem, Index );
```

We can now close all of our open loops.

```
                    } // Next Alpha Primitive
                } // End if alpha primitives
            } // Next Property Group
        } // Next Property Group
    } // Next Property Group
} // Next Light Group
```

At this point, all opaque polygons have been rendered and the hash table contains all of our alpha polygons. All that is left to do is to render the hash table in back to front order with a call to CScene::RenderSortedAlpha.

```
        // Render the alpha polygons (if any)
        if ( GetGameApp()->m_bSecondPassAlpha ) RenderSortedAlpha( );
}
```

CScene::AddAlphaSortItem

Each element in the hash table can be a pointer to a linked list of ASORT_ITEM structures. Therefore we must traverse through the linked list and insert polygons into the correct position such that larger distances are at the head of the list and smaller distances are stored towards the tail of the list.

```
void CScene::AddAlphaSortItem( ASORT_ITEM * pItem, ULONG HashIndex )
{
    float fDistance = pItem->Distance;

    // Attach this item to an element in the sort container hash table
    ASORT_ITEM * pIterator = m_pSortContainer[ HashIndex ], * pPrevious = NULL;

    // Anything in this list ?
    if ( !pIterator )
    {
        // Just add the item
        m_pSortContainer[ HashIndex ] = pItem;
        pItem->Next = NULL;

    } // End if no linked list
```

In the above code, we first store the distance in a local variable for easy access and readability. We then assign an ASORT_ITEM pointer called ‘pIterator’ the value of the pointer stored in the hash table at the passed index. We also create a pointer called ‘pPrevious’ and initially set this to NULL. Next we test to see if the pIterator pointer is NULL, and if so, then it means no items have been added to this index in the hash table. In that case, we can simply assign the hash table index the address of the passed item pointer. We then set the inserted ASORT_ITEM’s next member to NULL indicating that this item is the only item in the list and does not have other ASORT_ITEM structures attached.

If pIterator does not equal NULL then it means there must be at least one ASORT_ITEM structure already stored at this index. If this is the case, we need to traverse through the linked list until we find an ASORT_ITEM that has a smaller distance than the distance of the item we are trying to insert. We then insert the item just before it by attaching the item’s Next pointer to the pIterator.

```
    else
    {
        // Add it to the linked list in the correct position
        for(pIterator=m_pSortContainer[HashIndex]; pIterator; pIterator = pIterator->Next)
        {
            if ( pIterator->Distance <= fDistance )
            {
```

If this is the first iteration of the loop, the pPrevious pointer will be NULL and we should start off by inserting the item at the head of the list. We do this by assigning the item’s ‘Next’ pointer to the

current head of the list, and assigning the hash table pointer (which currently points to the old head of the list) to the new item such that our new item is now the new head of the list. The previous head of the list is now the second item in the list and is pointed to by our newly inserted item's Next pointer.

```
                if ( !pPrevious )
                {
                    pItem->Next = m_pSortContainer[ HashIndex ];
                    m_pSortContainer[ HashIndex ] = pItem;
                } // End if set as head
```

If this is not the first iteration of the loop, we need to insert the ASORT_ITEM item such that its Next pointer points to the pIterator (the ASORT_ITEM with a smaller distance). The previous item in the linked list (the pIterator from the previous iteration of the loop) will have its next pointer set to point at the newly inserted item. The passed ASORT_ITEM is inserted in the list like so: **pPrevious→pItem→pIterator**.

```
               else
               {
                     pItem->Next = pIterator;
                     pPrevious->Next = pItem;

               } // End if insert item

               break;

             } // End if we should insert here
```

For each iteration of the loop we store the current pIterator in the pPrevious pointer so that in the next iteration of the loop we have access to it. This allows us to insert the item between pPrevious and pIterator as was shown above.

```
             // Store previous item
             pPrevious = pIterator;

         } // Next Item in linked list
```

If we get here then pIterator is NULL. This means that we did not find an ASORT_ITEM in the linked list with a smaller distance value than the ASORT_ITEM we are trying to insert. Thus, we need to add our item to the end of the list.

```
         // If we reached the end of the list, just place it there
         if ( !pIterator ) pPrevious->Next = pItem;

    } // End if linked list already here
}
```

CScene::RenderSortedAlpha

The final function called at the very bottom of the CScene::Render function is CScene::RenderSortedAlpha. This function traverses the hash table from bottom to top rendering the linked lists of polygons stored at each index in the hash table.

```
void CScene::RenderSortedAlpha( )
{
    long          i, j;
    CLightGroup * pLightGroup       = NULL;
    ULONG       * pLightList        = NULL;
    CLightGroup * pLastLightGroup   = NULL;
    long          nLastMaterial      = -2;
    long          nLastTexture       = -2;
    ASORT_ITEM  * pItem = NULL, * pNextItem = NULL;
    LPDIRECT3DINDEXBUFFER9 pLastIndices = NULL;

    // Enable alpha blending and disable Z-Writing (to help reduce visible errors)
    m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_ZWRITEENABLE, FALSE );
```

The first thing we do is enable alpha blending and disable Z buffer writes. Then we loop through each element in the hash table starting at the end of the array and working our way back to hash table element 0.

```
    // Render back to front order
    for ( i = SORT_HASH_SIZE - 1; i >= 0; i-- )
    {
```

We now loop through each item stored at element ‘i’ in the hash table. All we are doing here is traversing the linked list stored at the hash table index. If there are no polygons (ASORT_ITEM’s) stored here, pItem will be NULL and the loop will immediately exit.

```
        for ( pItem = m_pSortContainer[i]; pItem; pItem = pNextItem )
        {
            // Collect light group
            pLightGroup = pItem->LightGroup;

            // Only set lights / vertex buffer if lightgroup is different.
            if ( pLastLightGroup != pLightGroup )
            {
                pLightList  = pLightGroup->m_pLightList;
                for ( j = m_nReservedLights; j < m_nLightLimit; j++ )
                {
                    if ( (j - m_nReservedLights) >= (pLightGroup->m_nLightCount ) )
                    {
                        // Disable any light sources which should not be active
                        m_pD3DDevice->LightEnable( j, FALSE );
                    }
                    else
                    {
                       // Set this light as active
                      m_pD3DDevice->SetLight(j,
```

```
                                        &m_pLightList[pLightList[j-m_nReservedLights]]);
                                         m_pD3DDevice->LightEnable( j, TRUE );
                    }

               } // Next Light
```

For each item stored in the linked list at the current hash table index, the above code retrieves a pointer to the light group stored. If the light group is different from the light group used to render an alpha polygon in a previous iteration of this loop, then we need to setup the device's lights using the lights in the current light group.

Once we have set the lights up for this new light group we bind the light group vertex buffer to device stream 0 and assign the local variable pointer pLastLightGroup the current light group pointer. This way in the next iteration of the loop, when processing the next item in the linked list, if the polygon stored there uses the same light group, we do not set up all of the same light states unnecessarily.

```
      // Set vertex stream
      m_pD3DDevice->SetStreamSource( 0, pLightGroup->m_pVertexBuffer,
                                     0, sizeof(CVertex));

      pLastLightGroup = pLightGroup;

   } // End switching light group
```

Now we extract the texture index for this polygon from the current item in the list and set the texture in stage 0. If the texture index is –1 then the polygon is not textured, and the texture stage will have its texture set to NULL. Once again, we store the current texture index in the nLastTexture local variable so that we do not unnecessarily set a texture if it has been set by the previous item in the list.

```
    // Set Properties
    if ( nLastTexture != pItem->TextureIndex )
    {
        if ( pItem->TextureIndex >= 0 )
        {
            m_pD3DDevice->SetTexture( 0, m_pTextureList[ pItem->TextureIndex ] );
        }
        else
        {
            m_pD3DDevice->SetTexture( 0, NULL );
        }

        nLastTexture = pItem->TextureIndex;

    } // End if different
```

Next we extract the material index from the ASORT_ITEM. If it is different from a material already set, we set the material using the material index from the current item we are processing and store the material index in the nLastMaterial local variable to avoid needlessly setting the material in the next iteration.

```
            // Set material
            if ( nLastMaterial != pItem->MaterialIndex )
            {
                 m_pD3DDevice->SetMaterial( &m_pMaterialList[ pItem->MaterialIndex ] );
                 nLastMaterial = pItem->MaterialIndex;
            }
```

Now we set the index buffer used by this item, and again employ the technique of remembering which one is set so that we do not needlessly set the index buffer later. Note that an index buffer may contain many alpha polygons so there is a very good chance that multiple triangles from the same index buffer will be rendered.

```
            // Set Indices
            if ( pLastIndices != pItem->Indices)
            {
                 m_pD3DDevice->SetIndices( pItem->Indices );
                 pLastIndices = pItem->Indices;
            }
```

Using the remaining information in the current ASORT_ITEM we are processing, we render the polygon stored there.

```
            // Render this primitive
            m_pD3DDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, pItem->BaseVertex,
                                                pItem->VertexStart, pItem->VertexCount,
                                                pItem->IndexStart, 1);
```

Finally, we store a pointer to the next item in the list and delete the current item from the linked list and from memory. This is because the ASORT_ITEM structures are allocated, added to the hash table and deallocated each frame. Note that this is certainly not the most efficient or memory friendly approach, so you are free to replace the dynamic memory allocations with a more robust system of your own.

The reason we store what the current item's Next pointer points to is because at this time, our only way to navigate the rest of the list is from this pointer. When the current item is deleted, we would lose the ability to access the rest of the linked list. So instead, we simply unhook the current item from the head of the list remembering the next item in the list, delete the current head of the list, and make the Next item in the list the new head of the list.

```
            // Clean up after ourselves
            pNextItem = pItem->Next;
            delete pItem;
            m_pSortContainer[i] = pNextItem;

        } // Next item in linked list for this entry
    } // Next hash table entry

    // Reset render states
    m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );
    m_pD3DDevice->SetRenderState( D3DRS_ZWRITEENABLE, TRUE );
}
```

After we have rendered all the polygons in our hash table, we re-enable Z writing and disable alpha blending.

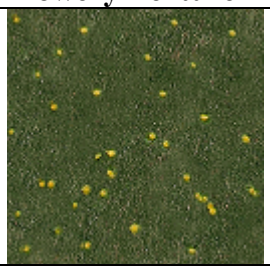
In this project we learned that whether we are using vertex alpha, texture alpha or both, we need to render our alpha polygons *after* we have rendered our opaque polygons. If the alpha polygons can never overlap from the viewer's perspective then this is all we really need to do. However, in most cases the alpha polygons need to be sorted and rendered in a back-to-front order to correctly blend properly. We also discussed various ways to generate the sorted alpha list and finally settled on the use of a hash table. This is a good solution in many situations but not our only option. The quicksort is also a very common technique when you have a lot of polygons to sort. Whether using the quicksort or hash table method of sorting, if the polygons dynamically move about throughout your world, then you will need to make sure you re-calculate and update the center points of these polygons. Using the center point sorting technique is not a perfect solution as it will generate incorrect results in certain situations (such as 'intersecting polygons'). These problems can only be eliminated using sub-division techniques such as BSP trees. However, this technique of sorting polygons will be ample for the vast majority of applications and is extremely fast when used with the hash table since no actual sorting/swapping needs to take place.

Lab Project 7.4: Alpha Surfaces

Terrain **texture splatting** is a technique that allows us to render a terrain that is built from a number of tiling texture layers. Texture splatting was used to generate the terrain in the commercial title Drakken II™ for the Sony Playstation™.

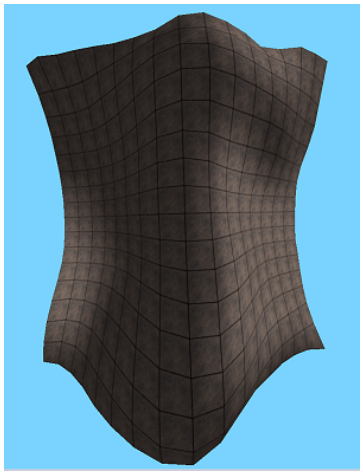
Unlike our previous terrain demos where we had to create a large terrain texture and map it to the four corners of the terrain, no such specific texture is needed when rendering a terrain using texture splatting. When using texture splatting we define a terrain as having a number of layers, where each layer is a structure that contains the tileable layer texture used by that layer (such as a grass texture or a mud texture for example). In Lab Project 7.4 our terrain is constructed using three layers: a rock layer, a grass layer, and a flower layer.

These form layers 0, 1, and 2 of the terrain respectively. Below you can see the three textures that belong to each of our three terrain layers. These textures are ones that are used to render the layer.

Layer 0 **Concrete Texture**	**Layer 1** **Grass Texture**	**Layer 2** **Flowery Texture**
		

These three textures are tileable textures which means that we can set the texture coordinates of the terrain vertices outside the [0, 1] range to make them tile over the terrain multiple times without noticing the seams. If you look at the screen shot above, you can see the concrete, grass, and flower textures all being used to render the terrain. Each layer texture is tiled multiple times across the terrain and we can see that the three textures blend flawlessly into one another without the need for us to create a final terrain texture in a paint package. We will simply feed our tileable texture layers into the system and let the texture splatting algorithm determine the final per-pixel color of our terrain. We will see in a moment how it is the alpha map assigned to each layer that determines how the three layers blend with each other.

Even if we had just a single layer, the fact that we are tiling the texture across the terrain at a high frequency (much like we did with our detail map in Chapter Six), means that we no longer need a detail map. We can set the texture coordinates of the mesh such that a layer texture is repeated multiple times across a single quad if we wanted to.



The image on the left shows a terrain constructed from a 17 by 17 height map. Hopefully you will remember from previous chapters that this means we will have a terrain mesh with 17x17 vertices forming 16x16 quads. In this image, we are looking the down the negative Y axis of the world from above. The mesh for the height map could be calculated as:

```
CVertex *pVertex = pTerrainVertexArray;

for (int Y = 0; Y < TerrainHeight; Y++)
for (int X = 0; X< TerrainWidth; X++)
{
        pVertex.x = X * Scale.x;
        pVertex.y = pHeightMap[Y*TerrainWidth + HeightHeight]*Scale.y;
       pVertex.z = Y * Scale.z;
        pVertex.tu = X;
        pVertex.tv = Y;
}
```

In the above code, pTerrainVertexArray is assumed to be an array of 17x17 vertices. Scale is a 3D vector where each component is used to scale the loop variables and the height map value into an arbitrary world space size. Notice that for each texture coordinate we just use the offset of the vertex into the height map. Thus, the vertex created from height map pixel (2,2) will have UV coordinates of (2,2) also. This means that the top left quad of the terrain will have textures coordinates (0,0) top left, (1,0) top right, (1,0) bottom left and (1,1) bottom right. We know that this will map the entire texture to the first quad. The second quad (moving along horizontally to right from the previous quad) will have texture coordinates of (1,0) top left, (2,0) top right, (1,1) bottom left and (2,1) bottom right. This means (provided the default texture coordinate addressing mode is enabled where the sampler states D3DSAMP_ADDRESSU and D3DSAMP_ADDRESSV are set to D3DTADDRESS_WRAP) texture coordinates outside the 0.0 to 1.0 range will tile repeatedly. So in the above code, each quad in our terrain will have the entire tileable texture mapped to it. You can see that the concrete texture has been tiled 16 times horizontally and 16 times vertically across the terrain in the picture. The concrete texture in this example is 512x512 in size, so we have an ample amount of per-pixel detail per quad. In fact, even in this simple mesh example where we are discussing a very small mesh of 17x17 vertices, if we wanted to get the same detail using a single texture and draped over the terrain, we would need a texture that was 8704x8704 in size. If that was a 32 bit texture it would consume 289 megabytes of video memory -- most graphics cards do not have that much memory. A more realistic terrain mesh size, even a modest 128x128 quad terrain constructed from a 129x129 height map, would consist of (128*128) 16,384 quads. Using tiling we can map a 512 texture per terrain quad. To get the same amount of texture detail using a single texture, the texture would need to have (128*512) * (128*512) = 4,294,967,296 pixels. At 32-bit, that is 4 bytes per pixel, which means the texture would consume 16 gigabytes.

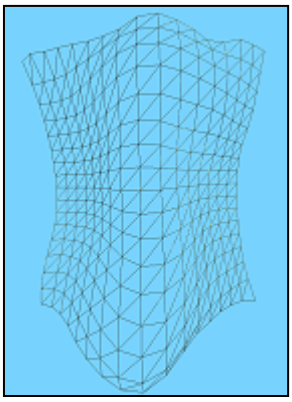
So clearly, using tileable textures and repeating them across the terrain affords us texture resolution that would otherwise not be possible. We will see later on that although the mesh is constructed such that its texture coordinates map a texture per quad, we can assign each layer a texture matrix used whilst rendering. This matrix can be used to scale the texture coordinates on the fly to generate arbitrary mappings. In this demo, we will set up each layer's texture matrix to be a scaling matrix that scales the U and V texture coordinates of each vertex by 0.5. So instead of the top left quad of the terrain having UV coordinates (0,0) (1,0) (0,1) (1,1) , when the quad is rendered its texture coordinates will be scaled to (0,0) (0.5,0) (0,0.5) (0.5,0.5). In this case only the top left quadrant of the texture is mapped to the top left texture and we have a mapping of one texture per four quads of terrain.

Tiling a texture across a terrain allows us to easily tweak the texture resolution. However it would now seem that we have lost the ability to color our terrain such that some areas are grass, some areas are mud, etc. With our old method of generating one big texture and laying it over the terrain, we could control which parts of our terrain were grassy and which were rocky simply by coloring the terrain manually. However, because the resolution of the texture was so low at the quad level, it made it difficult to place things such as roads or rivers in a precise way because one texel might be mapped to several quads. If we are going to use tileable generic textures and repeat them over the terrain, how can we control which parts of the terrain will use the grass texture and which parts of the terrain will use the rock texture? It is precisely this problem that the texture splatting technique is designed to overcome.

Texture Splatting 101

Before we discuss texture splatting and the various implementation considerations we must take into account, we will first discuss the concept at a very high level. Texture splatting is a very simply technique in theory but does tend to have a slightly more complex implementation in order to make it more efficient.

We need a terrain height map…



In our previous terrain demonstrations, we used a height map to generate the terrain vertex data. The same will be true in this project as well. To simplify the discussion on texture splatting for the time being, we will use a small 16x16 quad terrain to show the various examples (see image on left). In the actual demo we will use a 129x129 height map to generate a 128x128 quad terrain.

Our first task is to set up a number of layers. For the sake of this discussion we will say that a layer is nothing more than a structure that holds a base texture, such as grass, mud or rock etc., and a texture matrix used to describe how the texture assigned to that layer is tiled. With this approach a grass layer texture may be tiled across the terrain at a frequency of 2x2 quads per texture and a rock layer might be tiled at a frequency of 8x8 quads per texture. As a texture matrix can also contain rotational transformations, we could also apply a rotation to a layer's texture coordinates such that one layer might be tiled horizontally whilst another layer tiles diagonally. This tiling independence helps break up the uniform looking tiling of the texture layers.

At this time, let us imagine our terrain storing an array of vertices describing the terrain mesh, and an array of layers. A layer describes the textures stored at each layer and the texture matrix used to transform the UV coordinates of the layer.

```
class CTerrain
{
        IDirect3DVertexBuffer9    *pMeshData;       // Vertex buffer containing vertices
        DWORD                      VertexCount;     // How many vertices
        CTerrainLayer             *pLayers;         // Array of texture layers
        CTerrainSplat             *pSplats;         // Array of texture splats (one for each layer)
};

class CTerrainLayer
{
        IDirect3DTexture9 *     pLayerTexture;   // grass, mud , water etc
        IDirect3DTexture9 *     pBlendTexture;   // Texture used to blend the layer texture
        D3DXMATRIX              mtxTexMatrix;    // Controls scaling or rotations of mesh UVs
};
```

This is only for the sake of discussion. We know from our previous terrain applications that the terrain class will contain a lot more than two arrays. The above structures are purely to keep the discussion of texture splatting simple at this point. The CTerrain class structure above also contains a pointer to an array of CTerrainSplat objects, so we will now discuss what exactly a texture splat is.

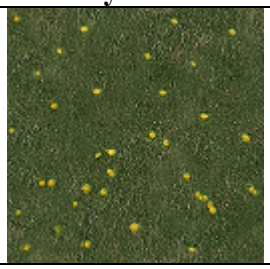
What is a Texture Splat?

In our simple example, if a terrain has three layers, where each layer describes a tileable texture that will be used to paint the terrain, then the terrain will also contain three splats. So there is one splat for each layer. A simple splat structure is shown below.

```
class CTerrainSplat
{
        IDirect3DIndexBuffer9   * pSplatFaces;
        DWORD                     IndexCount;
        DWORD                     PrimitiveCount;
};
```

A splat is a collection of indexed quads from the terrain vertex buffer that use the texture assigned to the matching layer. In other words, if we have three layers, then our above terrain will have an array of three texture splat objects; one for each layer containing the faces used by that layer. Therefore, if TextureLayer[0] is a grass texture, then TextureSplat[0] will contain an index buffer that describes all the quads in the terrain that use that grass texture – these will be rendered together when we render TextureLayer[0]. So at its most basic level, a texture splat is just an index buffer that contains a collection of terrain faces that use the matching layer texture.

Each splat will also contain an alpha texture. When we render the terrain, we start off at the base layer, and render each layer one at a time using the matching splat index buffer. For each layer, we set the texture in stage 0 and the matching splat alpha texture in stage 1. The color of each fragment is taken from the base texture assigned to stage 0, and the alpha value of each fragment is sampled from the alpha blend texture set in stage 1. Once we have the stages set for a given layer, we render the index buffer belonging to the corresponding terrain splat. So if we are rendering CTerrainLayer[1] for example, we assign the layer one texture to stage 0, then assign the blend texture belonging to CTerrainSplat[1] to texture stage 1. Then we render the index buffer belong to CTerrainSplat[1]. The alpha map assigned to a splat/layer controls how transparent or opaque the layer's texture is at a given point on the terrain.

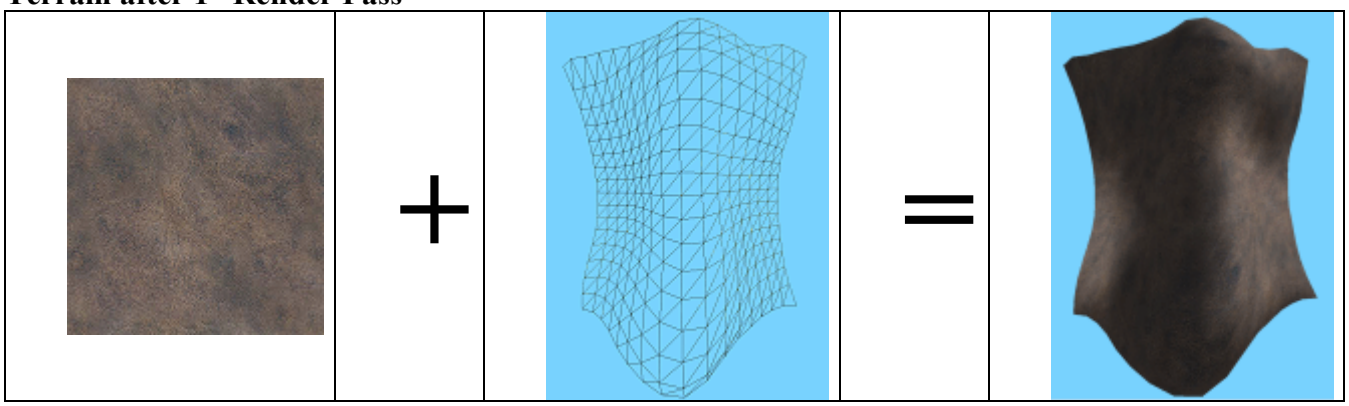
Layer 0 Rock Texture	Layer 1 Grass Texture	Layer 2 Flowery Texture
		

In our simple example, we will have an array of three CTerrainSplat objects, one for each texture layer depicted above. For simplicity, let us imagine that each splat object has an identical index buffer that contains all of the indices required to completely render every quad of the terrain. If we wanted to render all of the layers without using the alpha texture stored at each splat level, we could do the following:

```
pDevice->SetStreamSource ( 0 , pMeshData , 0 , sizeof (CVertex) );
for (int I = 0; I < NumberOfLayers;  I++)
{
     pDevice->SetTexture     ( 0 ,  &pLayers[I].pLayerTexture );
     pDevice->SetIndices     ( pSplats[I].pSplatFaces );
     pDevice->SetTransform ( D3DTS_TEXTURE0, &pLayers->mtxTexMatrix );
     pDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, 0, VertexCount,0,
                                                  pSplats[I]->PrimitiveCount );
}
```

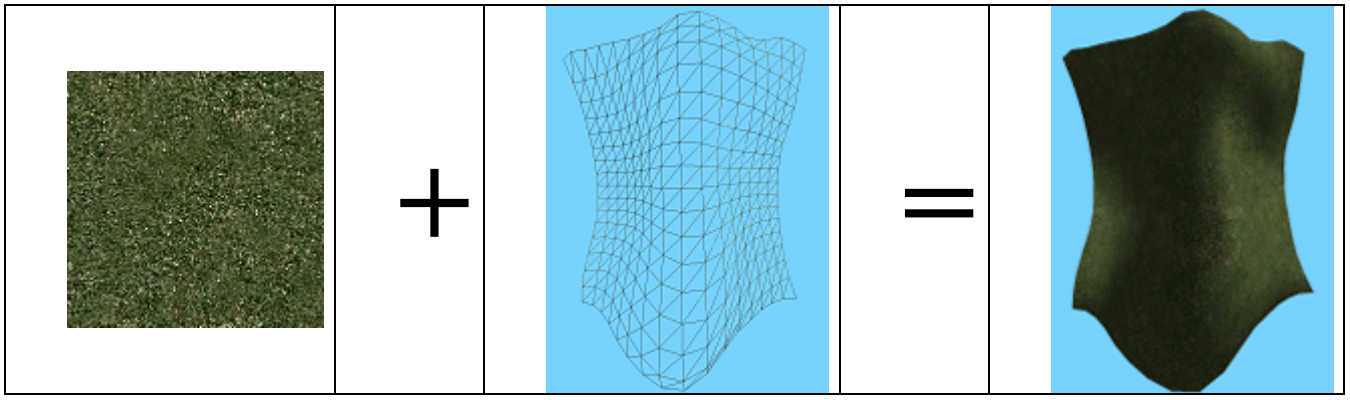
Assuming that the depth comparison function has been left at its default, where pixels are not rejected and are rendered if the Z value is less than or equal to a value already stored in the Z-Buffer, we can see that after the first iteration of the above loop we will have rendered the first layer (the rock layer). The frame buffer would contain the following image:

Terrain after 1st Render Pass



The resulting terrain is pretty much as expected. You could say at this point then, that we have rendered the first layer of the terrain, where the index buffer contains the faces used by the splat for layer 0. Since, in our current example, each splat index buffer contains an exact copy of all the terrain faces, we can see that in the second iteration of the loop, we render the entire terrain again using the second layer's texture (grass). Since we are not yet using the alpha textures, this will completely overwrite the terrain faces rendered in the frame buffer previously with the rock texture:

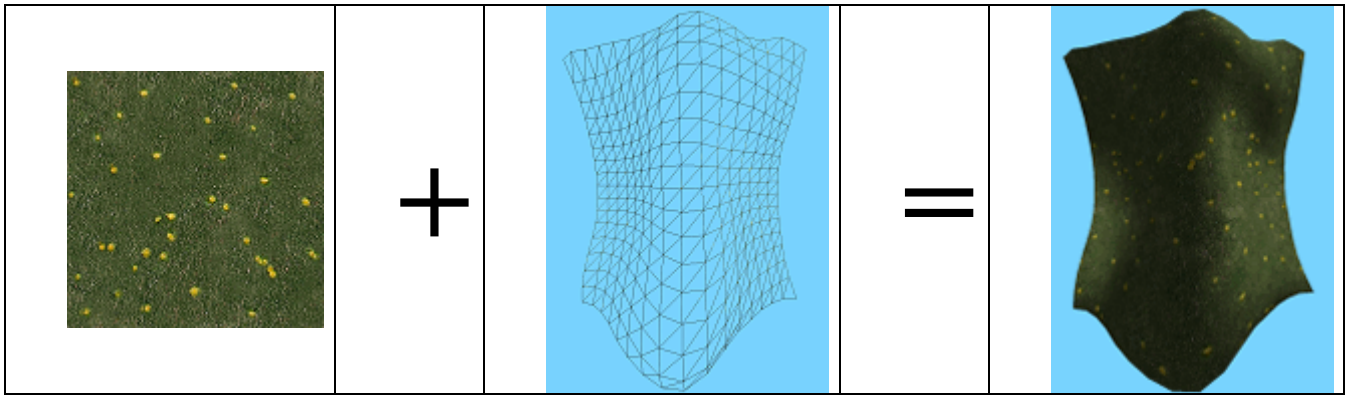
Terrain after 2nd Render Pass



We now have the same terrain tiled with the grass texture. No sign of the rock texture remains. Do not worry about how useless this seems to be at the moment. Right now we just want to understand that a texture splat terrain is rendered by looping through each layer and rendering either all or a portion of the terrain in a given number of passes.

Finally, in the third iteration of the loop in the above code, we render the third splat buffer. This will overwrite all of the grassy faces shown above with another copy of the terrain using the third layer texture:

Terrain after 3rd Render Pass



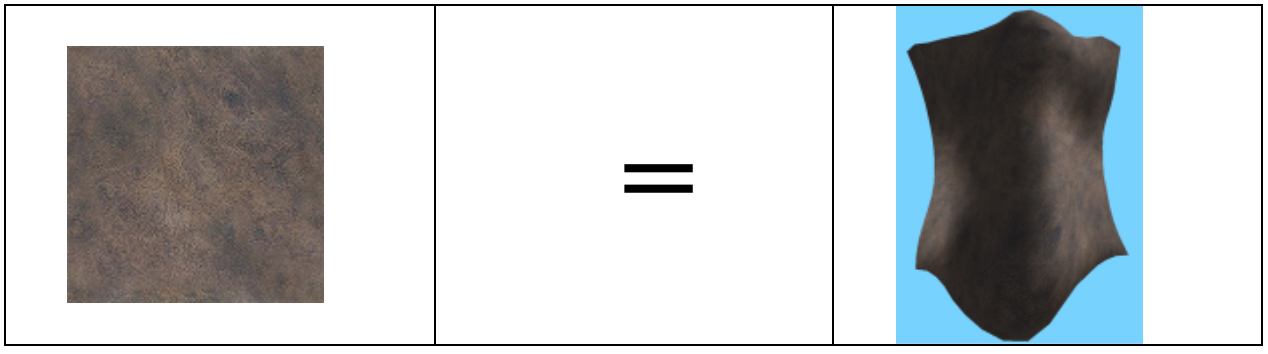
The Alpha Texture

So we build up our terrain by rendering a number of texture splats. In the above example, where each splat had identical index buffers, each successive splat that was rendered completely overwrote the previous splats/layers that had already been rendered. True, we could simply enable alpha blending to allow us to blend each layer such that we could see the splats that had previously been rendered, but this would still render a terrain where all layer textures are being mapped to all quads of the terrain, but this is not what we want. We want to have some areas specified as grassy and others as rocky for example. Furthermore, we also want smooth transitions where the terrain slowly blends from a grassy area into a rocky area without sharp edges between textures.

We do this by enabling alpha blending and rendering each splat using an alpha texture. This texture will describe how transparent the splat should be a particular point on the terrain. If the alpha value at that point is transparent, the contents of the frame buffer will show through. If the alpha value at that point is opaque, then anything currently in the frame buffer at that point will be overwritten just as in the above examples. If the alpha value is partially transparent at that point on the terrain, then the layer texture currently being rendered will be partially blended with the current contents of the frame buffer. This allows us to smoothly blend from one region to another. Let us see an example.

The Base Layer – The Rock Layer (No Alpha Map)

The base layer, Layer[0] will never have an alpha map because it never needs to be blended with anything underneath it. This is because it is rendered first and there is nothing currently in the frame buffer. Therefore, in our simple example, after we have rendered the first layer/splat the frame buffer will contain the mesh shown next. This is exactly the same result as when we rendered the mesh in the above examples. The texture is completely opaque.



The Second Layer – The Grass Layer

Every layer except the base layer needs a texture and an alpha map. This alpha map is not tiled like the layer's base map. It is draped over the entire terrain much like the terrain base texture used in previous

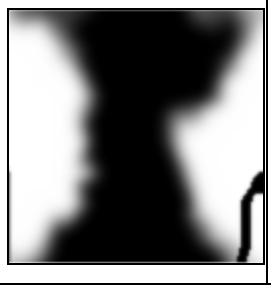
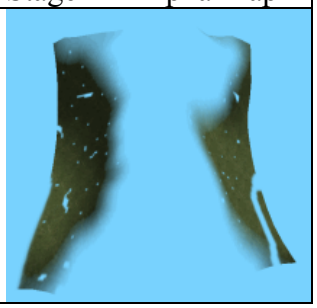
applications. This means that the terrain vertices will need a second set of texture coordinates where the vertex at the top left corner of the terrain has a UV coordinate of (0,0) and the vertex at the bottom right corner of the terrain has a UV coordinate of (1,1).

The alpha map in our example will not be a pure alpha surface although it certainly can be if the hardware supports it. For compatibility reasons, we have used a 16-bit A4R4G4B4 texture where the RGB components are not used and only the alpha value stored in each texel is used by the pipeline. This texture is set in the second texture stage. The stages are configured to sample the alpha from the texture and use that as the alpha value output from the pipeline to blend the terrain into the frame buffer.

In our application, we created the alpha map by simply creating a 24-bit RGB texture in a paint package. We then selected a white spray gun tool and adjusted its opacity settings such that the longer you hold the spray gun down over a given pixel the whiter it becomes. The white pixels will become the totally opaque pixels, whilst the black pixels will be totally transparent. Every pixel is a shade of gray between black and white which means for any pixel of the image, all the RGB components are the same. For example, for half intensity gray, the RGB components would be (128,128,128). When we load this image into our application, we will only need an alpha value, so we can use any one of the color components to get this result and copy it into the alpha component of the alpha texture we are creating.

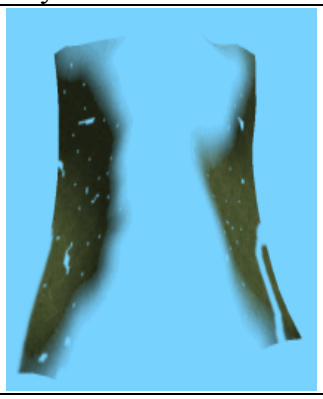
For example, as you will see later in our application, we load the image into memory, create a DirectX texture with an alpha channel, and copy only the blue component of each pixel in the source image into the destination texture's corresponding pixel alpha component. After we have filled the alpha components of each pixel in our layer alpha texture, we can discard the original 24-bit grayscale image from memory since it was only needed to fill out the alpha channel of our texture surface. The important point here is that we can easily design our alpha maps as normal 24-bit color images and extract the blue component (or red or green) and copy it into the alpha component of the blend texture. This blend texture will be assigned to the second texture stage and have its alpha sampled.

In the next image we see the base map (the grass texture) used by layer 1 and its alpha map. Again, white pixels represent total opacity and black pixels represent total transparency; pixels of any other shade represent a blending of some degree. The rightmost image shows that if we rendered this layer by itself into an empty (blue) frame buffer, the alpha map controls the blending of the terrain with the frame buffer.

Layer 1 Base Texture		**Layer 1** Alpha Map		**Rendered Blended Terrain** Stage 0 – Base Texture Stage 1 – Alpha Map
	+		=	

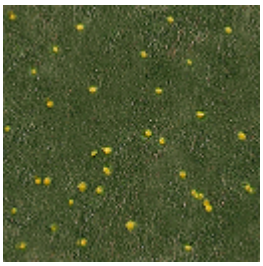
The above images show the output of rendering splat 1. In this example we are rendering the terrain using layer 1's texture assigned to stage 0. The color operations set for that stage sample the color from the base texture and the alpha from the alpha texture assigned to stage 1. The color sampled from the first stage and the alpha sampled from the second stage, are used to blend with the frame buffer to create the final image in the empty frame buffer. Things should be starting to make a little more sense now.

What about if we render layer 0 first and follow it with the second layer shown above? Remember that layer 0 is never alpha blended and never has an alpha blend map assigned to it. Rather than our grass splat mesh being blended with the blue frame buffer, it will instead be blended with the splat mesh from layer 1 which already in the frame buffer. The results are shown below:

Render Layer 0 No Alpha Map		**Render Layer 1** Using Alpha Map Layer 1		**Frame Buffer** Blended Result
	+		=	

The 3rd Layer – The Flower Texture

The 3rd layer of our terrain consists of a grassy/flower texture and an alpha map which dots the flower texture about the terrain. Below we see the base texture and its alpha map, and the result of rendering the splat mesh into an empty frame buffer using our texture/alpha texture stage states that sample the color from the base map and the alpha from the alpha map:

Layer 2 Base Texture		**Layer 2** Alpha Map		**Rendered Blended Terrain** Stage 0 – Base Texture Stage 1 – Alpha Map
	+		=	

It should be pretty easy to imagine what the final terrain would look like after we have alpha blended our third splat mesh into the frame buffer (on top of the previous two splats). Below we see the final results of blending the texture from layer 3 into the frame buffer after the first two splats were rendered.

Frame Buffer after first two layers have been rendered		**Render Layer 2** Using alpha map as shown above.		**Frame Buffer** Blended Result consisting of all 3 layers.
	+		=	

And there we have it; the basic process of texture splatting. The pseudo-code to render a 3-layer terrain is:

```
// Stage 0 Coloring : Modulate vertex color and texture color
pDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE );
pDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
pDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );

// stage 0 alpha : Just set up flow to next stage
pDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1 );
pDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_CURRENT );

// stage 1 coloring : Output stage 0 texture color unaltered by this stage
pDevice->SetTextureStageState( 1, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
pDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_CURRENT );

// stage 1 alpha : Output alpha sample from texture assigned to this stage.
pDevice->SetTextureStageState( 1, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1 );
pDevice->SetTextureStageState( 1, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );

pDevice->SetStreamSource ( 0 , pMeshData , 0 , sizeof (CVertex) );

for (int I=0; I< NumberOfLayers; I++)
{
    pDevice->SetTexture     ( 0 ,  pLayers[I].pLayerTexture );    // Set the layer texture
    pDevice->SetTexture     ( 1 ,  pLayers [I].pBlendTexture );   // Set the  alpha texture
    pDevice->SetIndices     ( pSplats[I].pSplatFaces );           // Set the splats indices
    pDevice->SetTransform ( D3DTS_TEXTURE0, &pLayers->mtxTexMatrix );
    pDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, 0, VertexCount,0, // Render the splat
                                                        pSplats[I]->PrimitiveCount );
}
```

This is all very simple in theory, but we will need to discuss some optimizations. Firstly, in the above example we were rendering the entire terrain three times because all the splats had index buffers that contained every face in the terrain. This amount of overdraw will become prohibitive on larger terrains and is something we must minimize. Our solution is straightforward enough. If a layer has completely transparent quads, then the splat for that layer need not render those quads at all. We should only have faces stored in its index buffer that contribute to the frame buffer in some way. Furthermore, if a layer has completely opaque quads, then they will completely overdraw any quads in the same position belonging to lower level splats. So these quads can also be removed from the lower level splat index buffers in a pre-process. The basic process will be as follows:

Load our blend maps, one for each layer
For Each Blend Map
** For Each Pixel in Blend Map**
**   For (This Layer+1 to all the layers above this one)**

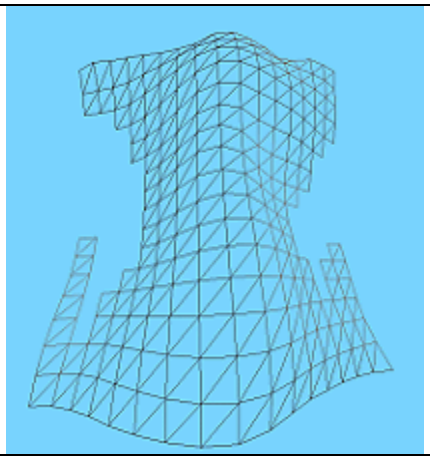
If the matching pixel in any of the above layers blend maps is 255 (completely opaque) set the pixel in the current blend map to zero because it is completely obscured

end for This Layer+1
End for each Pixel in Blend Map
End for each Blend Map

The above algorithm is run before the indices for each splat level are calculated. At the end, a blend map that has totally opaque pixels will set the corresponding pixels in the lower layer blend maps to zero. This is beneficial because when we fill the index buffers of each splat level, we will loop through the blend map of each layer and generate a quad only if the four corresponding pixels in the blend map are not all zero. Therefore, by canceling out pixels in lower level blend maps, we are also effecting the way the splat's index buffer will be filled -- it will only contains quads that are not completely transparent.

This does however raise an interesting question regarding the base layer. Since the base layer has no alpha map and is always considered opaque, how do we remove the quads from this layer? Well, although the base layer will not need an alpha map during rendering, during the creation of the splats for each layer we do temporarily create a blend map for the base layer that starts off with every pixel opaque. We do this by allocating the memory and filling each pixel with a bright white color. Next, we execute the above algorithm. At the end of the loop, this temporary blend map will have every pixel that is completely occluded by the higher level layer blend maps set to black. We then build the base layer splat index buffer using this blend map just as we do with the other higher level layers. Once the splats for each layer have been generated, this temporary base layer blend map can be discarded.

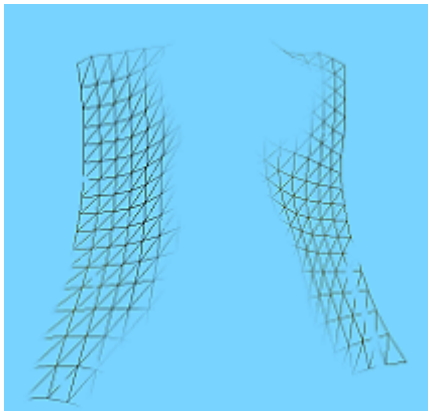
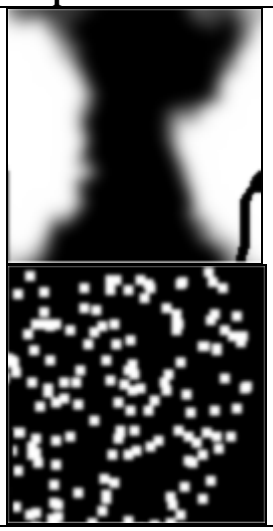
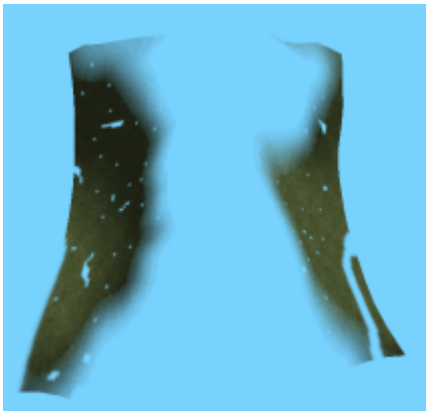
Below we can see what the splat for the base layer would look like after either completely transparent quads or quads completely obscured by higher level splats have been removed from consideration when building the splats index buffer.

Base Layer Splat	Layer 1 Alpha Map	Rendered Result
		

As you can see, we are no longer rendering the complete terrain. The base layer is now is only a subset of the actual terrain quad set. The quads that have not been added to this base layer splat have been rejected because when rendering, they would have been overdrawn by higher level splats. You can see by looking at the alpha map for the layer above the base layer, that where the pixels in the alpha map are white, the quads belonging to the lower layer splat are removed. This is because the second layer's quads will be fully opaque at these points.

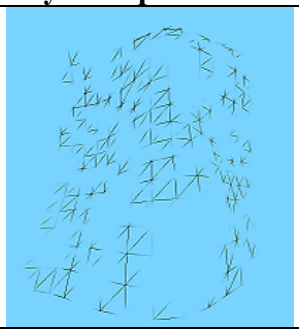
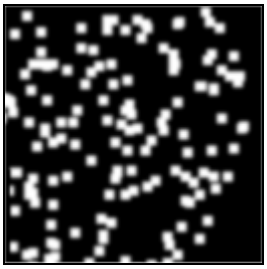
You can also see that where the layer above has black pixels in the alpha map, the current layer will not be overdrawn and will need to show through, so we leave the quads in place. We only remove quads from a splat if the corresponding pixels in the occluding layer's alpha map are totally white (full intensity). You will notice by looking at the alpha map of the 2nd layer that there are also pixels that are not full intensity white. These represent quads that will be partially transparent to some degree, and as such will not totally occlude the quads in the base layer. It is the quads that correspond to these pixels that will allow us to blend smoothly from one layer to the next.

Using this simple quad elimination method, the second layer splat now consists of fewer quads than before. It now contains only a subset of terrain quads that are not transparent and not completely overdrawn by a higher splat level. It should also start to become clear exactly why we are storing a unique index buffer for each splat. After this quad occlusion testing is done, each splat will contain a unique list of face in its index buffer describing only the quads used by that splat. Remember however that there is only one copy of the terrain vertex data and it is stored in a vertex buffer in the terrain class. Each splat is only an index buffer that describes a subset of that vertex buffer for a given layer.

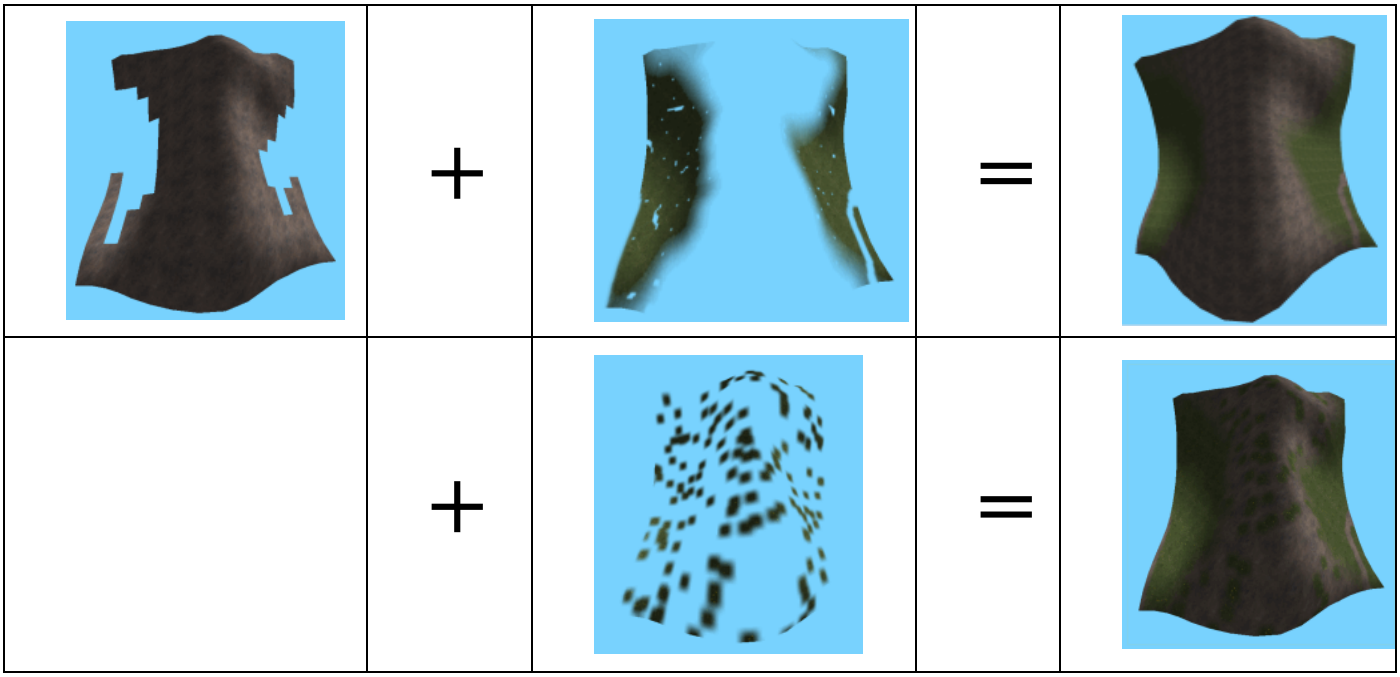
Layer 1 Splat	**Layer 1 Alpha Map** **Layer 2 Alpha Map**	**Rendered Result**
		

In the above diagram we see the alpha map for layer 1 and layer 2 as they both influence the mesh that will be created and rendered. We can see by looking at the wire frame mesh, that when the splat is built for layer 1 we are only generating quads where the layer map for that layer is not completely black. We also see the alpha map for layer 2 here since it will be used at occlusion testing time to drill down through the layers (as discussed above) and set any alpha map pixels to black in lower layer alpha maps that are occluded by its white pixels. Although the Layer1 map shown above is in its state before this occlusion testing has taken place, the rendered result shows that white pixels in the Layer2 map have created some transparent holes in the Layer1 map. There was not enough to actually remove any quads in this case because the Layer2 alpha map is too sporadic. There are just not enough opaque masses in the alpha map to occlude any underlying quad. If you look at the rendered result however, you will see that there are now little transparent bits in the grassy regions that are not there in that layer's alpha map. This is where the above layer (the flower layer) completely overdraws the grass (and rock) layers.

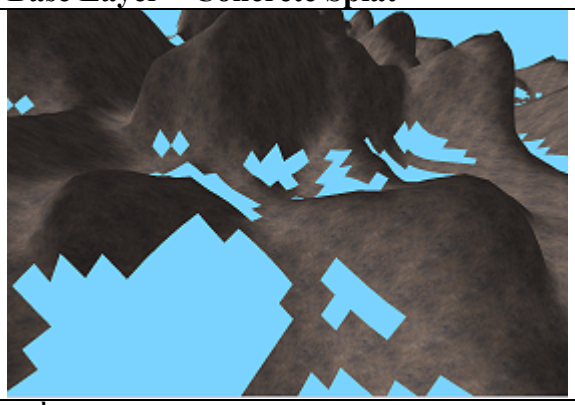
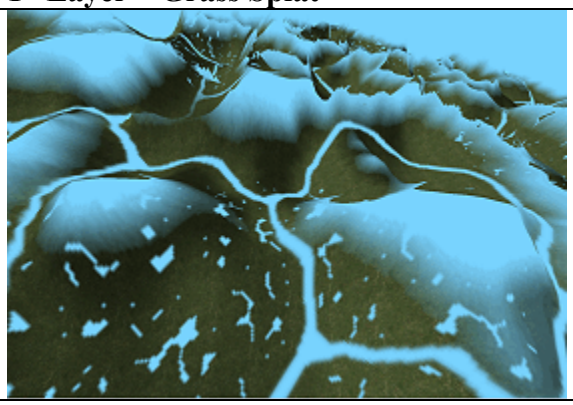
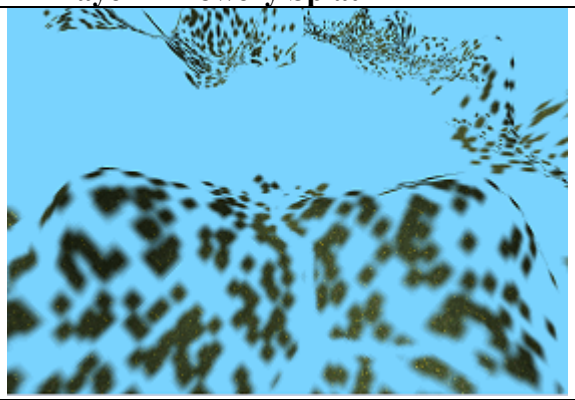
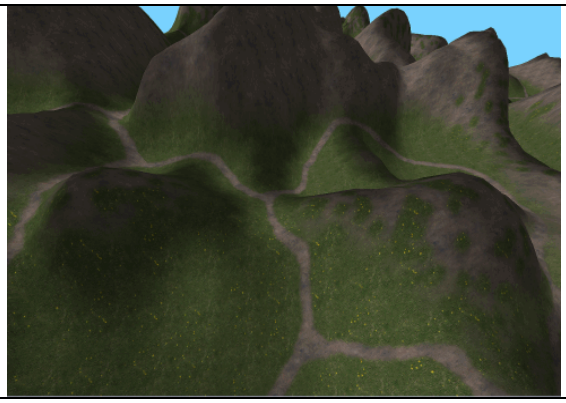
The following images show the splat created for the third and topmost layer, rendered using our flower texture to dot some random looking foliage about the terrain. Notice how the spaces in the splat where there are missing quads match up exactly with the larger dark regions of the layer's alpha map. Because this is the top layer splat, its quads cannot be occluded by anything above it. Therefore, the rendered result and the mesh created are exactly what we would expect from looking at the associated alpha map.

Layer 2 Splat	**Layer 2 Alpha Map**	**Rendered Result**
		

When we render these three layers using three passes we get the correct results, because quads are only missing from a given splat if they would be overdrawn anyway:



Hopefully, at this point we are all clear as to what exactly texture splatting is and how it works in theory. The following screen shots show the larger terrain that we will use in Lab Project 7.4 to give us a better idea of how the individual layers are used to construct the final terrain. In our demo, the base layer is rock and the second layer is grass. As a result, we can draw black lines on the grass layer alpha map so that the base layer shows through. This creates roads and trails on the terrain.

Base Layer – Concrete Splat		1st Layer – Grass Splat	
	+		
2nd Layer – Flowery Splat		Rendered Result	
	=		

Implementation Considerations

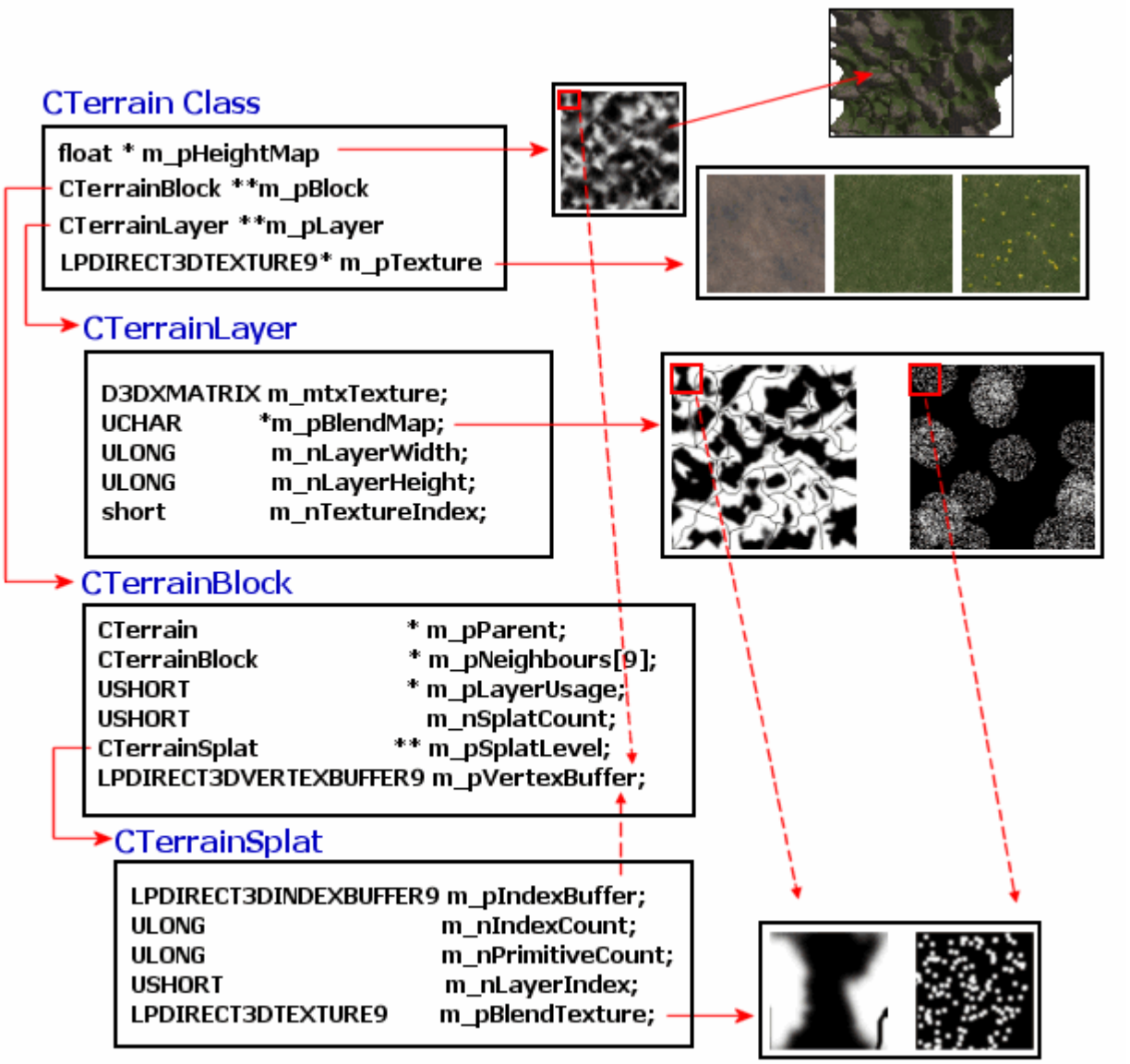
It is now time to take a look at how our application will implement texture splatting. At first you might think that it would be much simpler to have the terrain store three layer objects describing the base texture and the alpha map for that layer and three splat levels containing the index buffers for each layer. However we learned in previous lessons that if we have a large terrain, we want to break it down into sub-meshes so that we are not sending too many vertices into the pipeline at one time. In our previous demos, each terrain block was really just a vertex buffer describing a terrain submesh. We created a terrain from a 257x257 height map which meant that the overall terrain consisted of 257x257 vertices (and 256x256 quads). We broke this down into sub-blocks where each block stored a 17x17 capacity vertex buffer and each terrain block was a 16x16 quad sub-terrain. The same technique will be employed in this demo although we will be using a 129x129 height map this time to create a 128x128 quad terrain. Each terrain block will still be a 16x16 quad submesh however. This means we have to slightly modify the way that we store our data. It is now the individual terrain blocks that will store the three splat levels for that block. Each splat index buffer will describe only the quads for a given layer that belong to that terrain block. Rendering the terrain can then be done as follows.

```
For Each Block of the Terrain
    For Each Layer
        If (Block.SplatLevel[layer] exists) Render ( Block.Splat(Layer) )
    End for Layer
End for Block
```

Notice that we check that a splat exists for a given terrain block before we try to render it. This is because we can think of each terrain block as storing three splats, where the splats represent a 16x16 quad area for a given layer. If a particular layer has totally transparent pixels in that region of the terrain, then no quads are generated for that block layer. Therefore, it is possible that although our terrain is using three layers in total, not all terrain blocks will have all three splats defined. Imagine for example that part of our terrain was fully rock textured and this section was one of our terrain blocks. This terrain block would only have one splat level because it does not use quads from the other two layers. The bottom line is that not all terrain blocks need to have a splat for each layer.

When we create a height map for our terrain and the various alpha maps for each layer, these alpha maps will be defined for the entire terrain. This is, they will be draped over the full terrain. However, when we load an alpha map for a given layer, we will carve it up into pieces and assign a piece of the alpha map to each terrain block. This means, in a simple example where each terrain block uses all three layers of the terrain, each block will store three splats, one for each layer. In that case each splat will consist of an index buffer for the terrain block and an alpha texture for that terrain block. Rendering each terrain block would consist of rendering all three of its splats, where each splat uses its assigned alpha texture during rendering. Once we have carved a layer's alpha map into terrain block sized alpha textures, the original alpha map can be discarded from memory as it will not be needed in the rendering process.

Let us have a look at the data structures we will use to represent the terrain in this project: the CTerrain class, the CTerrainBlock class, the CTerrainLayer class, and the CTerrainSplat class. In the following diagram we see the basic relationships between the four classes as well as some of the more important member variables. We also see some examples of data that might be stored in some of those members. The CTerrain class has a great deal more to it than is shown in this diagram but what we are looking at here is just the new stuff. The rest of the terrain members are the same as our previous terrain demos. The CTerrain class contains an array of CTerrainLayer objects as well as the height map data used to build each terrain block vertex buffer. The CTerrain class also contains an array of base textures used by each layer. In our application there are three base textures: rock, grass, and flower. Each of the three CTerrainLayer objects in the CTerrain layer array contains an index into this texture array describing which texture the layer uses and a texture matrix describing the transformation for the first set of texture coordinates in the layer. This allows us to control the frequency and angle at which the base texture is tiled across the terrain. The CTerrainLayer object also contains a blend map, describing the alpha information used for the entire terrain. Each layer map, except the base layer alpha map, is loaded from a file and all layer maps are 1024x1024 pixels. The base layer map is generated programmatically by starting off with a layer map that is completely white (opaque), and then setting pixels that are occluded by higher level layer maps to black (transparent).



Looking at the above design, we can see that the terrain class builds itself from the height map as a series of CTerrainBlocks. Each terrain block contains 16x16 quads and an array of CTerrainSplat objects. A terrain block that contains all three layers will have three CTerrainSplat objects. Each CTerrainSplat object belonging to a terrain block contains an index buffer with the block quads and a blend texture. The CTerrainBlock blend texture is a texture with an alpha channel. It contains only a section of the corresponding layer's alpha map that applies to that block. In the diagram we see the blend textures that would be created for the second and third splat objects of the top left 16x16 terrain block of the terrain. If all blocks use all layers, then each terrain block will contain three splat objects and each splat object will contain an alpha texture that contains the section of the alpha map that applied to that block. These CTerrainSplat blend textures are created at application startup by dividing the layer maps stored in the CTerrainLayer object into block sized alpha maps. Once the alpha maps have been created for all splats, the larger parent blend maps can be discarded. The only information

the CTerrainLayer object contains (during rendering) is the index of the base texture used by this layer. Notice that each terrain block contains a USHORT array called m_pSplatUsage. If our terrain has three layers, then each terrain block will have an m_pLayerUsage with as many elements as the number of layers (3 in our example). When we create each terrain block, we will fill this array such that if an element of this array is non-zero, then the terrain block has a splat for the corresponding layer. In other words, if a terrain block has splats for only layers 0 and 2, the splat usage array will be [1,0,1] for that terrain block. This allows us to check quickly during rendering whether a terrain block uses a certain layer. If so, then the corresponding splat of that terrain is rendered. The CTerrainBlock class also stores a pointer to the parent terrain class and its nine neighboring terrain blocks. This information is handy to keep around as we will see later. When we discuss the source code, you should refer back to the above diagram to remember how the four objects are related.

The next implementation detail we must discuss is how to set up the various parameters for the terrain and each of its layers. Although in previous projects we have simply hard-coded such data items as how many blocks our terrain get carved into, and how many quads a terrain block consists of, we now have this information and much more that will need to be fed into our texture splatting terrain engine. For example, we need to set the base texture and the alpha map texture for each layer. We need to store the parameters that allow us to setup a texture matrix for each layer controlling how the base texture is tiled. So rather than hard-coding this information, we have decided in this project to store this information in an .ini file. Our application will read the information from this file during terrain setup. This allows us to tweak the operating parameters of our engine without the need to touch any code or have to recompile our application. This technique is something that has been used for many years across all sorts of applications. Many Windows applications for example use .ini files to store their settings. If you performed a search for .ini files on your computer you would probably find a great many listed in the results box. Whilst .ini files have been superseded by the window registry for storing permanent application settings, the registry is generally not a good place to store information that we wish the user to be able to change easily. An inexperienced user digging about in the windows registry can cause damage to the operating system. However, if the user incorrectly alters an .ini file, all he or she can do is prevent the application that relies on the .ini file setting from running correctly (or at all).

Before we discuss the splatting source code, let us take a quick look at the .ini file and discuss the sort of information our application will expect to be stored there. We will examine how an .ini file is laid out and briefly touch on the Windows API functions at our disposal for extracting settings from such a file.

Initialization files 101

.ini files are text files containing key/value pairs. They can be created in Windows Notepad or any plain text editor. They are laid out in such a way that we can use Windows API functions to extract the information from these files easily, without needing to explicitly read the file ourselves. Associated settings can be grouped into ‘sections’. A section is a block of settings surrounded by square brackets. All the settings are said to belong to a given section until another section head is encountered in the file.

A simple .ini file is shown next. It has two sections that we have called 'MySection1' and 'MySection2'. MySection1 has three keys: Name, NickName and Age. MySection2 contains two keys called Gender and Status. Each of these keys is assigned a value, and it is these values that the application will extract using the section name and the key name to describe value we wish to extract in the file.

```
;--------------------------------------------------------------------
; My Example INI File
; -------------------------------------------------------------------

[My Section1]
Name =      Gary Simmons
NickName    = [GI]-Gaz
Age  =      32

[My Section2]
Gender =    Male
Status =    Single
```

This simple file shows us all we need to know. Lines that start with a semicolon are comments. We have total freedom to call the different sections of our file whatever we like. There are also no limits in terms of how many sections we can store and no limit on the number of keys a section can contain. This is really a flexible way to store data parameters such that our users can redefine the default operating parameters of our applications.

Extracting data from .ini files is very straightforward since the Windows API provides a number of easy to use functions. The two that we are interested in are shown below. They are used for extracting string and integer values respectively. First we look at how to extract an integer value from our file.

```
UINT GetPrivateProfileInt
(
  LPCTSTR lpAppName,  // address of section name
  LPCTSTR lpKeyName,  // address of key name
  INT nDefault,       // return value if key name is not found
  LPCTSTR lpFileName  // address of initialization filename
);
```

The first parameter should be a pointer to a string containing the name of the section. For example, we would specify "My Section1" if we were trying to extract the Age value from our test file. The second parameter is the name of the key. In our example we have only one numerical value assigned to the key 'Age' in the first section of our file. Therefore, we would pass in "Age" as the second parameter. The third parameter allows us to specify a default return value if the key we are looking for does not exist in the file. This enables our application to act responsibly and handle missing keys with sensible default values. For example, we might specify a value of 18 here in case the Age key is not found. The last parameter is a string containing the name of the 'ini' file we wish to extract the information from. If this parameter does not contain a full path to the file, the system searches for the file in the Windows directory.

Next we see how this function can be used to extract the 'Age' value stored in the 'MySection1' section of our .ini file above. In this example, we are expecting the .ini file to be in the 'Data' directory of the current working directory. If the value is not found for whatever reason, a default value of 18 will be returned.

```
char IniFileName[MAXPATH];
UINT AgeValue;

// Get the current working directory and append the sub folder and file name
GetCurrentDirectory( MAX_PATH, IniFileName );
strcat( IniFileName, "\\Data\\Level1.ini" );

// Extract the Age value from the 'My Section1' section and store it in the AgeValue variable
AgeValue = GetPrivateProfileInt ("My Section1" , "Age" , 18 , IniFileName );
```

The actual file reading is done for us behind the scenes and we are done.

The second function our application will want to use extracts strings from .ini files. We will need this because we store the texture names for each splat layer in the .ini file. This allows us to apply different textures to the terrain simply by adjusting the texture names in the file.

```
DWORD GetPrivateProfileString
(
  LPCTSTR lpAppName,        // points to section name
  LPCTSTR lpKeyName,        // points to key name
  LPCTSTR lpDefault,        // points to default string
  LPTSTR lpReturnedString,  // points to destination buffer
  DWORD nSize,              // size of destination buffer
  LPCTSTR lpFileName        // points to initialization filename
);
```

Most of the parameters should be fairly obvious given our prior discussion. The only ones to note are the fourth and fifth parameters. The fourth parameter is where we pass in a pointer to a destination buffer that will receive the extracted string. The fifth parameter specifies the size of our destination buffer so that the function will only copy over data that will fit within the buffer and not overflow the buffer accidentally.

The example code below extracts the 'Name' string from 'My Section1' and the 'Gender' string from 'My Section2' in our test file:

```
char UserName[1024];
char UserGender[5];

// Get the current working directory and append the sub folder and file name
GetCurrentDirectory( MAX_PATH, IniFileName );
strcat( IniFileName, "\\Data\\Level1.ini" );
```

```
// Extract Name value ( 'Gary Simmons' )
GetPrivateProfileString ( "My Section1" , "Name"   , " Unkown" , UserName   , 1024 , IniFileName );

// Extract Gender Value ( 'Male' )
GetPrivateProfileString ( "My Section2" , "Gender" , " Unkown" , UserGender , 5      , IniFileName );
```

Now that we see how easy it is to extract values and strings from an .ini file, let us have a quick look at the .ini file we created for Lab Project 7.4. We will examine it one section at a time, starting with the first. This is the [General] section used to hold settings that apply to the entire scope of the application.

```
[General]
Name         = Test Terrain
Desc         = This terrain is designed to test the texture splatting technique.
Heightmap    = Heightmap.raw
Scale        = 190.0, 10.0, 190.0
TerrainSize  = 129, 129
BlendTexRatio = 8
BlockSize    = 17, 17
LayerCount   = 3
```

The 'Name' and 'Description' strings are there in case we want to display information about what the application is and what it does. The 'Heightmap' key contains the name of the heightmap file that our application should use to build the terrain. The 'Scale' key assigns a scale vector to control how large the terrain will be in the world in the X, Y, and Z dimensions respectively. The 'TerrainSize' key describes the width and height of the height map. This is important to know since the height map is stored in a .raw file (which contains raw pixel data and no width or height information). The 'BlendTexRatio' value is our way of specifying the size that our layer blend maps will be. A value of 8 means that each layer's alpha map file will be 8 times larger in each dimension. If the terrain if 128x128 quads, the alpha map files for each layer will be expected to be (128*8) x (128*8) = 1024x1024 pixels. This ratio means that each quad of our terrain will use an 8x8 square of pixels from each alpha map, allowing us to have smooth blending even within a single quad. If you were to lower this value to 4 for example, the layer alpha maps would need to be 512x512 pixels and each quad would be mapped to a 4x4 block of pixels in each alpha map. The 'BlockSize' key contains the size that we would like each terrain sub block to be in terms of vertices. In our example, we are stating that the terrain should be broken down into sub meshes that are 17x17 vertices in size (16x16 quads). Finally, the 'Layer' key in the [General] section describes how many layers this application will use. Our demo will use the three layers discussed earlier. The [General] section is a mandatory section for our application, so it should not be removed.

The next section in the .ini file is the [Textures] section. This is also a mandatory section. It describes how many textures our application will use and the filenames of each layer's base texture.

```
[Textures]
TextureCount  = 3
Texture[0]    = Textures\leath03.jpg
Texture[1]    = Textures\grass.tga
Texture[2]    = Textures\grassmeadow.tga
```

The next section in the file describes the base layer of the terrain and is called the [Base Layer] section. The base layer does not have an alpha map file that needs to be loaded but we will still specify values such as scale and rotation for the texture matrix.

```
[Base Layer]
TextureIndex  = 0
Scale         = 0.5, 0.5
Translation   = 0.0 , 0.0
Rotation      = 0
```

This section describes the texture to use for this layer, which is index 0. If we cross reference this with the [Textures] section we can see that it is a texture called leath03.jpg. We also specify a texture coordinate scaling factor to control the tiling frequency. In the above example tiling will occur such that one texture repeat is mapped to 2x2 quads (4 quads). We can also specify a 2D translation vector to apply an offset to the texture coordinates and a rotation value which should be specified as a single float in degrees. In this demo, we are not using either value and as such they could be omitted as our application will choose the same default values. In fact, in our actual .ini file these two keys are not listed in the [Base Layer] section. We show them here simply to make you aware that these keys exist

After the [Base Layer] section, there are a number of [Layer N] sections, where 'N' is the number of the layer the settings belong to. For example, in our demo that has a base layer and two additional layers, there will be a [Layer 1] section and a [Layer 2] section, making three layers in all. The [Layer N] sections are similar to the [Base Layer] section except that they also specify a file name for the layer blend alpha map.

```
[Layer 1]
LayerMap      = Layer1.png
TextureIndex  = 1
Scale         = 0.5, 0.5
Rotation      = 45
 [Layer 2]
LayerMap      = Layer2.png
TextureIndex  = 2
Scale         = 0.5, 0.5
Rotation      = 45
```

We see that [Layer 1] uses texture index 1 as its base texture. When we cross reference this index with the [Textures] section we see that this is the 'grass.tga' file. This layer also uses a blend map called 'Layer2.jpg' which our application will load and carve up to create the individual blend maps for each splat for that layer. The grass texture uses the same (0.5, 0.5) scaling that allows for one texture to be mapped to four quads. We also rotated the texture 45 degrees so that the grass tiles diagonally across the terrain.

Source Walkthrough

Now that we know what our application will find in the .ini file it is time to step through the code and see how all of the things we have discussed can be implemented. The texture splatting code is all part of the CTerrain class contained in the file CTerrain.cpp.

Using the splat terrain from the CGameApp class is no different from using our previous terrain class. The CGameApp class still has a CTerrain member variable and in the CGameApp::BuildObjects function we again instruct the terrain to construct itself. In our previous terrain projects, the CGameApp::BuildObjects function called the CTerrain::LoadTerrain function passing in the filename of the height map. However, since all of this information is now contained in the .ini file, we will pass the name (with path) of the .ini file instead.

```
GetCurrentDirectory( MAX_PATH, IniPath );
strcat( IniPath, "\\Data\\Level1.ini" );

// Build the terrain data
m_Terrain.SetD3DDevice( m_pD3DDevice, HardwareTnL );
if ( !m_Terrain.LoadTerrain( IniPath )) return false;
```

The terrain is rendered from the CGameApp::FrameAdvance function as in our prior terrain demos. We use a single call to the CTerrain::Render function as shown below.

```
// Begin Scene Rendering
m_pD3DDevice->BeginScene();

// Render player mesh FIRST because terrain may render alpha components
m_Player.Render( m_pD3DDevice );

// Reset our world matrix (player sets it)
m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_mtxIdentity );

// Render our terrain objects
m_Terrain.Render( m_pCamera );

// End Scene Rendering
m_pD3DDevice->EndScene();
```

As we can see, nothing has really changed with respect to using and rendering the terrain. All of our changes are found within the terrain class itself.

The CTerrain Class

Let us first look at the CTerrain.h header file and examine the class declaration. We will also look at the supporting classes, such as the CTerrainBlock class (a sub mesh of the terrain), the CTerrainLayer

class (a base texture and alpha map) and the CTerrainSplat class (the index buffer for each layer for each splat) along the way.

Below we see the member variables of the CTerrain class. Many of these will be familiar from previous terrain implementations. We will not list the member functions here but will cover them as we call them from our code.

The first four variables store the information used to build the terrain mesh. These are a pointer to the height map data, the width and height of the height map, and the scale that we will use to create a world space vertex from an image space height map pixel. The values for each of these four variables are extracted from the .ini file.

```
class CTerrain
{
    private:

    D3DXVECTOR3    m_vecScale;         // Amount to scale the terrain meshes
    float         *m_pHeightMap;       // The physical heightmap data loaded
    ULONG          m_nHeightMapWidth;  // Width of the 2D heightmap data
    ULONG          m_nHeightMapHeight; // Height of the 2D heightmap data
```

The next group of variables stores information about how many vertices are going to be assigned to each terrain block and how many quads are contained in each block. We also track the number of terrain blocks the master terrain is broken into by storing how many blocks wide and high the terrain is. This is information that will once again be either directly extracted from the .ini file or calculated from information extracted from the .ini file.

```
    ULONG          m_nBlockWidth;      // Width of an individual terrain block
    ULONG          m_nBlockHeight;     // Height of an individual terrain block
    ULONG          m_nQuadsWide;       // Stores the number of quads per block
    ULONG          m_nQuadsHigh;       // Stores the number of quads per block
    USHORT         m_nBlocksWide;      // Number of blocks wide
    USHORT         m_nBlocksHigh;      // Number of blocks high
```

The next variable describes the size of the layer maps with relation to the terrain dimensions. This is another value extracted directly from the .ini file. We use a value of 8 in this demo. This means that a terrain constructed from 128 rows of quads with 128 quads in each row will have 1024x1024 layer maps. Thus an 8x8 group of texels in each layer map will be mapped to a single terrain quad. Here we use a single blend texture ratio for all layers, but you could alter the code so that each layer could store its own blend texture ratio.

```
    USHORT    m_nBlendTexRatio;   // Number of blend map texels to map to each terrain quad
```

The terrain class needs to store a pointer to an array of CTerrainBlocks pointers. Each terrain block is a subset of the terrain. The terrain block contains a vertex buffer with that subset's vertices and an array of CTerrainSplat objects. There is a splat for each layer used by the terrain block. Each splat stores an index buffer with the quads for that terrain block that belong to its associated layer. The terrain class

also contains an array of CTerrainLayer objects. Each layer contains an index to a texture used by that layer and a pointer to a layer map.

```
    CTerrainBlock     **m_pBlock;           // Simple array of terrain block pointers
    ULONG               m_nBlockCount;      // Number of terrain blocks stored here
    CTerrainLayer     **m_pLayer;           // Simple array of layer pointers
    USHORT              m_nLayerCount;      // Number of layers stored here
```

Finally, the CTerrain class needs to hold an array of textures. Since we have three layers (rock, grass, flower), there will be three textures in our array. Each CTerrainLayer object contains a texture index into this array describing the texture used by that layer.

```
    LPDIRECT3DTEXTURE9* m_pTexture;         // Array of textures loaded for this terrain
    USHORT              m_nTextureCount;    // Number of textures loaded
};
```

The CTerrainLayer Class

The CTerrain class maintains an array of CTerrainLayer object pointers. There is one pointer for each layer that the terrain uses. In our project, this array will contain three pointers. The CTerrainLayer class itself has a very simple job to do during rendering. It contains a texture index that will be used to determine which base texture will be tiled across the terrain for the current layer and a texture matrix to determine how that tiling occurs. The matrix will be filled based on the values specified in the ini file for a given layer.

The CTerrainLayer has a temporary use during terrain initialization. Notice that the class contains a UCHAR pointer to a blend map. This is the blend map that has its name specified in the ini file. During terrain creation we load each layer map into the corresponding CTerrainLayer object where it will later be used to build the blend map textures stored in each terrain block. The actual blend textures stored for each terrain block will be a small subset of the entire alpha map. They will contain only the alpha pixels that are mapped to that terrain block region. Once the blend textures have been built for each terrain block, the blend maps stored in the CTerrainLayer class can be discarded. Refer back to the earlier diagram to remind yourself of the relationship between the CTerrain, the CTerrainBlocks, the CTerrainLayers and the CTerrainSplats.

```
class CTerrainLayer
{
     public:

     // Constructors & Destructors for This Class
     CTerrainLayer();
    ~CTerrainLayer();

     // Public Variables For This Class
     D3DXMATRIX          m_mtxTexture;     // The texture matrix applied to this layer
     UCHAR              *m_pBlendMap;      // The blend map data for this layer
     ULONG               m_nLayerWidth;    // Width of the layer alpha map
     ULONG               m_nLayerHeight;   // Height of the layer alpha map
```

```
    short               m_nTextureIndex;  // Index of the texture to use

    // Public Functions for This Class
    UCHAR   GetFilteredAlpha( ULONG x, ULONG z );
};
```

GetFilteredAlpha is used to return a filtered average for a given pixel in the alpha map. Earlier we discussed that once we have the blend maps loaded, we can drill down through the layer maps and if we find a pixel that is totally opaque, set all the pixels in the lower layer alpha maps to zero. This prevents quads being built for that layer at those spots. Under normal circumstances we will have bilinear filtering enabled in the texture stages when the alpha map texels are sampled, so we cannot simply test a single pixel to see if it is white or not. Although a pixel may be totally opaque in our alpha map, when it is sampled by the DirectX sampler unit with filtering enabled, adjacent texel colors will also be summed up and a weighted average returned. So an opaque pixel in our alpha map might be diluted by the sampler given the surrounding texels and the color returned may no longer be totally opaque. If we have removed all quads from the lower layers under these circumstances, then we will see holes in the terrain. Therefore, the GetFilteredAlpha function allows us to use a filtered average while we are testing for occluded pixels in lower layers to avoid the bilinear filtering concern that might remove quads inappropriately.

The CTerrainBlock Class

The CTerrain class maintains an array of CTerrainBlock objects. Each terrain block is a subset of the terrain. When we render the terrain, the terrain object will in turn render each of its terrain blocks. There are only two public functions in this class and both are called from the CTerrain object. The CTerrainBlock::GenerateBlock function is called from the CTerrain class when the terrain is being constructed. It is passed the X and Y offset into the parent terrain height map and the width and height of each terrain block. This allows the function to index into the height map correctly and generate its vertex buffer using only the pixels in the passed section of the height map. The CTerrainBlock::Render function is called from the CTerrain::Render function. It is called once for each block, for each layer which requests the CTerrainBlock to render its splat mesh for the layer passed. If there are three layers, each terrain block will have its render function called once for each layer.

```
class CTerrainBlock
{
public:

  // Constructors & Destructors for This Class
  CTerrainBlock();
  virtual ~CTerrainBlock();


  // Public Functions For This Class
  bool    GenerateBlock ( CTerrain * pParent, ULONG StartX, ULONG StartZ,
                          ULONG BlockWidth, ULONG BlockHeight );

  void    Render        ( LPDIRECT3DDEVICE9 pD3DDevice, USHORT LayerIndex );
```

The following member variables record the X and Y offsets into the parent terrain height map, the block vertex width and height, and the block quad width and height. The terrain block also stores a pointer to the parent terrain and an array of nine CTerrainBlock pointers. These are pointers to its neighboring blocks (N, NE, E, SE, S, SW, W, NW).

```
  // Public Variables for This Class
    ULONG                   m_nStartX;        // X Position in heightmap we start
    ULONG                   m_nStartZ;        // Z Position in heightmap we start
    ULONG                   m_nBlockWidth;    // Width of an individual terrain block
    ULONG                   m_nBlockHeight;   // Height of an individual terrain block
    ULONG                   m_nQuadsWide;     // Number of quads in this block
    ULONG                   m_nQuadsHigh;     // Number of quads in this block
    CTerrain              * m_pParent;        // Parent terrain pointer
    CTerrainBlock         * m_pNeighbours[9]; // Neighbour block pointers
```

The TerrainBlock object also stores an array of unsigned shorts. The array will be large enough to have an element for each layer. In our current example where we use three layers, this array will have three elements. If an element is zero in this array, then this terrain block does not have a splat for the corresponding layer. For example, if the terrain block had a zero only in m_pLayerUsage[1], then the terrain block only has quads to render for layers zero and two.

```
    USHORT                * m_pLayerUsage;     // Layer usage table
```

Each terrain block contains an array of CTerrainSplat objects. Each splat contains an index buffer used for rendering the quads that belong to the terrain for a given layer. This splat index buffer is used to specify indices into the parent CTerrainBlock vertex buffer. Therefore, when we render the terrain block at the splat level, it will always contain a vertex buffer with the (17x17) section of the terrain.

```
    USHORT                  m_nSplatCount;    // Number of splat levels stored
    CTerrainSplat        ** m_pSplatLevel;    // Actual splat levels stored
    LPDIRECT3DVERTEXBUFFER9 m_pVertexBuffer;  // Terrain blocks vertex buffer
```

Finally, there are a number of private functions in this class that will be called from the public CTerrainBlock::GenerateBlock function. These six functions are shown below.

```
private:
    // Private Functions for this Class
    bool    CountLayerUsage     ( );
    bool    GenerateSplats      ( );
    bool    GenerateSplatLevel  ( USHORT TerrainLayer );
    long    AddSplatLevel       ( USHORT Count );
    bool    GenerateBlendMaps   ( );
};
```

The CountLayerUsage function populates the m_pLayerUsage array so that we know if a given layer is used by this terrain block. The GenerateSplats function allocates a CTerrainSplat object for each layer used by this terrain block and populates the splat index buffer. The GenerateSplats function calls the AddSplatLevel function to allocate the memory for the m_pSplatLevel array. In our example, it will allocate the array to be large enough to hold pointers to three CTerrainSplat objects. It then loops

through each layer and calls the GenerateSplatLevel function to set the data for each splat object just created. Finally, the GenerateBlendMaps function builds the individual blend maps for each splat. It extracts the data from the global blend maps for each layer, and copies only the section that is relevant to this terrain block into the blend texture. The end result is that for a given terrain block, each splat has an index buffer that renders quads for the associated layer using its own blend map texture for that same associated layer. If a terrain has three layers, then a terrain block will have three splats. Each splat will have an index buffer and a blend map describing how to alpha blend this splat into the frame buffer. After the large global layer maps have been used to build the per-splat blend maps, the large layer maps can be discarded from memory as they will not be used for rendering.

The CTerrainSplat Class

Each CTerrainBlock object contains N CTerrainSplat objects where N is the number of layers used by the terrain. In our demo project, each terrain block will contain three CTerrainSplat objects. Each splat contains an index buffer that indexes into the parent terrain block vertex buffer

```
class CTerrainSplat
{
public:
    // Constructors & Destructors for This Class
    CTerrainSplat();
   ~CTerrainSplat();

    // Public Variables For This Class
    LPDIRECT3DINDEXBUFFER9  m_pIndexBuffer;     // Index buffer for rendering splat
    LPDIRECT3DTEXTURE9      m_pBlendTexture;    // Generated blend texture
    ULONG          m_nIndexCount;      // Pre-Calculated Number of indices for rendering
    ULONG          m_nPrimitiveCount;  // Pre-calculated number of primitives for rendering
    USHORT         m_nLayerIndex;      // Layer index used for this splat level
};
```

Each splat object contains a pointer to an index buffer that describes the quads for this splat as well as the corresponding variables describing how many indices are in the buffer and the number of primitives stored. Each splat object also contains a layer index describing which layer this splat belongs to. Remember that a single terrain block will have a splat for each layer. Finally, we also store the alpha map. Each alpha texture is a subset of the original layer's alpha map that contains only the pixels that are mapped to this terrain block. This is the texture that will be set in the second texture stage during rendering and the texture surface from which the alpha value for each fragment will be sampled.

It is now time to look at the code to some of these classes and see how we have implemented them. The first class we will look at is the new CTerrain class. We will look only at the functions that have been added for this demo. We will not examine the code to simple helper functions and constructors and destructors -- which simply set default values. If you wish to see the code for such functions please check the accompanying source code.

CTerrain::LoadTerrain

Our CGameApp class calls the CTerrain::LoadTerrain function passing in the name of the .ini file. This function is indirectly responsible for the entire creation process for the terrain. When it returns, the terrain will be completely built, along with all of its layers, terrain blocks, and splat levels.

The first thing this code must do is extract the name of the height map from the .ini file as shown below.

```
bool CTerrain::LoadTerrain( LPCTSTR DefFile )
{
    FILE  * pFile = NULL;
    char    Buffer  [1025], Section [100], Value[100], FileName[MAX_PATH];
    ULONG   i;

    // Cannot load if already allocated (must be explicitly released for reuse)
    if ( m_pBlock ) return false;

    // Must have an already set D3D Device
    if ( !m_pD3DDevice ) return false;

    // Read in the terrain definition values specified by the file
    strcpy( Section, "General" );
    GetPrivateProfileString( Section, "Heightmap", "", FileName, MAX_PATH - 1, DefFile );
```

The above code searches the [General] section of the ini file for the 'Heightmap' key, and if found, copies the name of the heightmap into the 'FileName' local array. If the key is not found, a default string of "" is used. This will eventually cause this function (and the application) to return an error when the height map texture cannot be found.

Next we extract the terrain scale. It is stored as a vector in the .ini file where all three values are on one line. While vectors are not directly supported by the .ini files, we can just read the line of text assigned to the Scale key and parse the string ourselves. The following code extracts the scale string into a local char array called Buffer. If the Scale key is not found, a default string of "1,1,1" will be used. We then use the 'sscanf' to scan the string and extract the values into the floating point components of our scale vector.

```
    GetPrivateProfileString( Section, "Scale", "1, 1, 1", Buffer, 1024, DefFile );
    sscanf( Buffer, "%g,%g,%g", &m_vecScale.x, &m_vecScale.y, &m_vecScale.z );
```

We now extract the size of the terrain so that we know how big the height map will be and how much memory to allocate for it. We search for the TerrainSize key in the [General] section of the .ini file and copy the line of text into the local buffer. If the terrain size is not specified then a default value of 257x257 will be assumed. Our height map is only 129x129 so we will specify this in the .ini file. Once we have extracted the string, we use sscanf to extract the two values into our m_nHeightMapWidth and m_nHeightMapHeight member variables.

```
    GetPrivateProfileString( Section, "TerrainSize", "257, 257", Buffer, 1024, DefFile );
```

```
    sscanf( Buffer, "%i,%i", &m_nHeightMapWidth, &m_nHeightMapHeight );
```

Next we extract the terrain block size and the blend texture ratio into the appropriate member variables. A default terrain block size of 17x17 vertices is used if no value is specified and a blend texture ratio of 1 is the default value. This means that the alpha map for each layer would be expected to be the same size as the height map (minus 1) and as such, we would only have a blend ratio of 1 blend texel per quad. This would not be very good. Our .ini file specifies a blend texture ratio of 8 so that we have an 8x8 block of alpha map texels mapped to a single quad of our terrain. As such, each alpha map will be expected to be 1024x1024 in size.

```
    GetPrivateProfileString( Section, "BlockSize", "17, 17", Buffer, 1024, DefFile );
    sscanf( Buffer, "%i,%i", &m_nBlockWidth, &m_nBlockHeight );

    GetPrivateProfileString( Section, "BlendTexRatio", "1", Buffer, 1024, DefFile );
    sscanf( Buffer, "%i", &m_nBlendTexRatio );
```

Now we calculate how many quads wide and high each terrain block will be. In our application this will result in terrain blocks of 16x16 quads.

```
    // Store secondary data
    m_nQuadsWide = m_nBlockWidth - 1;
    m_nQuadsHigh = m_nBlockHeight - 1;
```

We allocate the memory for heightmap and load it in using the file name just extracted from the .ini.

```
    // Attempt to allocate space for this heightmap information
    m_pHeightMap = new float[m_nHeightMapWidth * m_nHeightMapHeight];
    if (!m_pHeightMap) return false;

    // Build the heightmap path / filename
    strcpy( Buffer, DataPath );
    strcat( Buffer, FileName );

    // Open up the heightmap file
    pFile = _tfopen( Buffer, _T("rb") );
    if (!pFile) return false;

    // Read the heightmap data
    for ( i = 0; i < m_nHeightMapWidth * m_nHeightMapHeight; i++ )
    {
        UCHAR HeightValue;
        fread( &HeightValue, 1, 1, pFile );

        // Store it as floating point
        m_pHeightMap[i] = (float)HeightValue;

    } // Next Value

    // Finish up
    fclose( pFile );
```

At this point we have our height map data loaded. The next step is calling the CTerrain::FilterHeightMap function to apply a simple box filter to our height map to smooth it out a bit. Because we are using a smallish height map and are scaling it to create a large terrain, this can have the effect of making the terrain appear jagged since the different height values in the height map are integer values. The reason we apply a filter in the code rather than in a paint package is that we can filter using floating point numbers instead of integers. This allows us to blur the map without integer granularity restrictions. The FilterHeightMap function is not part of the main process, and removing it will not stop the application from running. You are encouraged to test it yourself both with and without this filter being applied.

```
    // Filter the heightmap data
    FilterHeightMap();
```

Now that our height map loaded and filtered, it is time to extract the texture names from the .ini file and load the images onto texture surfaces. The first thing we do is extract the 'TextureCount' value from the [Textures] section of the .ini file. If it is non-zero, then we allocate an array of IDirect3DTexture9 interface pointers.

```
    // Load in the texture data
    strcpy( Section, "Textures" );
    m_nTextureCount = GetPrivateProfileInt( Section, "TextureCount", 0,DefFile );
    if ( m_nTextureCount > 0 )
    {
        // Allocate space for specified textures
        m_pTexture = new LPDIRECT3DTEXTURE9[ m_nTextureCount ];
        if ( !m_pTexture ) return false;
```

We loop and extract the texture name from the [Textures] section of the .ini file. They are stored in the form 'Texture[N] = XXX' where XXX is the base texture for layer N.

```
        // Loop through and read in texture filenames etc.
        for ( i = 0; i < m_nTextureCount; ++i )
        {
            // Build the Key we are looking for 'Texture1' , 'Texture2' etc
            sprintf( Value, "Texture[%i]", i );

            // Retrieve the filename
            GetPrivateProfileString( Section, Value, "", FileName, MAX_PATH - 1, DefFile );
```

Once we have the base texture name for a given layer, we append it to the DataPath and load it using the D3DXCreateTextureFromFileEx function. We store the resulting texture interface pointer in the CTerrain::m_pTexture array.

```
            // Build the texture path / filename
            strcpy( Buffer, DataPath );
            strcat( Buffer, FileName );

            // Load it in (Ignore failure, it's not fatal)
            D3DXCreateTextureFromFileEx( m_pD3DDevice, Buffer, D3DX_DEFAULT, D3DX_DEFAULT,
                                         D3DX_DEFAULT, 0, m_fmtTexture, D3DPOOL_MANAGED,
```

```
                                        D3DX_DEFAULT, D3DX_DEFAULT, 0,
                                        NULL, NULL, &m_pTexture[i] );
        } // Next Texture
    } // If any textures
```

At this point we have our height map loaded and the base textures for each layer stored in our terrain texture array. We then call the CTerrain::GenerateLayers function to load the large global alpha maps for each layer and build each layer's texture matrix based on the values extracted from the ini file. We follow this with a call to CTerrain::GenerateTerrainBlocks to build each terrain block and its accompanying splat levels and blend maps.

```
    // Generate the terrain layer data
    if ( !GenerateLayers( DefFile ) ) return false;

    // Build the terrain blocks
    if ( !GenerateTerrainBlocks() ) return false;
```

When CTerrain::GenerateTerrainBlocks returns, the terrain, its terrain blocks, its layers and its splat levels will be fully initialized. At this point, the large layer alpha maps that have been loaded by the GenerateLayers method and they will have been copied into sub blend maps for each splat and can be freed from memory.

```
    // Erase the blend maps, they are no longer required
    for ( i = 0; i < m_nLayerCount; i++ )
    {
        if ( m_pLayer[i]->m_pBlendMap )
        { delete []m_pLayer[i]->m_pBlendMap; m_pLayer[i]->m_pBlendMap = NULL; }
    } // Next Layer

    // Success!!
    return true;
}
```

CTerrain::FilterHeightMap

The first helper function called from CTerrain::LoadTerrain is CTerrain::FilterHeightMap. It performs a blend on the height map to smooth out the steps between integer height values before they are scaled into world space. This is very much like using a blend/blur function in a paint package.

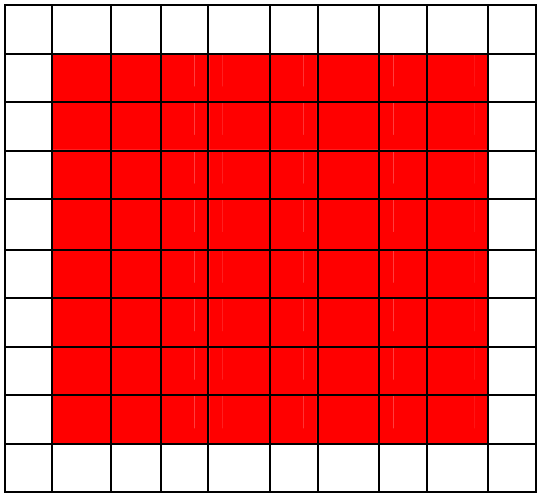
Our filter will loop through each pixel in the height map and will sum the color of a 9x9 block of pixels with the pixel in the middle. Once we have summed the nine colors together we will divide the final color by nine to produce an average pixel color for that region of the terrain as shown below.

200	0	100
9	87	99
128	238	45

Here we are processing a pixel that has a height value of 87. To calculate the new filtered height value for this pixel, we sum up the surrounding pixel heights:

200+0+100+9+87+99+128+238+45 / 9 = 100.666

Since we can only do this for pixels that actually have pixels surrounding them on all sides, we will start our filter pass at an offset of 1 pixel down and 1 pixel to the right from the top left corner of the image. We will filter pixels only contained within the rectangle [1, 1, ImageWidth-1, ImageHeight-1]. The following image shows the pixels we would filter (the red ones) in a 10x10 height map.



```
void CTerrain::FilterHeightMap( )
{
    ULONG x, z;
    float Value;

    // Validate requirements
    if (!m_pHeightMap) return;

    // Allocate the result
    float * pResult = new float[m_nHeightMapWidth * m_nHeightMapHeight];
    if (!pResult) return;

    // Copy over data to retain edges
    memcpy(pResult, m_pHeightMap, m_nHeightMapWidth * m_nHeightMapHeight * sizeof(float));
```

We allocate memory to store a copy of the height map and copy the height map data into this buffer. This allows us to alter the colors in the copy buffer while sampling the 9x9 block of colors for each pixel averaged from the original unaltered height map. Once done, the original height map will be released and the new filtered height map will become the terrain's actual height map. Copying into this buffer might seem strange at first since we are calculating the pixel values from this buffer one at a time anyway. However this approach makes sure that the four outside edges of the height map (which do not get recalculated in our filter loop) are copied into the new filtered height map buffer.

```
    // Loop through and filter values (simple box style filter)
    for ( z = 1; z < m_nHeightMapHeight - 1; ++z )
    {
        for ( x = 1; x < m_nHeightMapWidth - 1; ++x )
        {
            Value  = m_pHeightMap[ (x - 1) + (z - 1) * m_nHeightMapWidth ];
            Value += m_pHeightMap[ (x    ) + (z - 1) * m_nHeightMapWidth ];
            Value += m_pHeightMap[ (x + 1) + (z - 1) * m_nHeightMapWidth ];

            Value += m_pHeightMap[ (x - 1) + (z    ) * m_nHeightMapWidth ];
            Value += m_pHeightMap[ (x    ) + (z    ) * m_nHeightMapWidth ];
            Value += m_pHeightMap[ (x + 1) + (z    ) * m_nHeightMapWidth ];

            Value += m_pHeightMap[ (x - 1) + (z + 1) * m_nHeightMapWidth ];
            Value += m_pHeightMap[ (x    ) + (z + 1) * m_nHeightMapWidth ];
            Value += m_pHeightMap[ (x + 1) + (z + 1) * m_nHeightMapWidth ];

            // Store the result
            pResult[ x + z * m_nHeightMapWidth ] = Value / 9.0f;

        } // Next X
    } // Next Z
```

Once we have the filtered image data, we release the old height map data and assign CTerrain::m_pHeightMap to point at this new array.

```
    // Release the old array
    delete []m_pHeightMap;

    // Store the new one
    m_pHeightMap = pResult;
}
```

CTerrain::GenerateLayers

GenerateLayers is the second helper function called by CTerrain::LoadTerrain. Its job is to generate the terrain layers. This basically means that this function will allocate memory for N CTerrainLayer objects (where N is the number of layers specified in the .ini file). Each CTerrainLayer object that is allocated has its pointer added to the CTerrain::m_pLayer array. For each layer added we need to allocate the memory for the layer alpha map and load it from a file. We also need to extract the values from the .ini file to build the texture matrix for that layer. Finally, we will loop through each pixel of each layer map and test to see if a pixel in a layer is totally obscured by opaque pixels in higher level layer alpha maps. If so, then we set the occluded pixel value to 0 so that it will prevent a quad from being built in that location for that layer.

```
bool CTerrain::GenerateLayers( LPCTSTR DefFile )
{
    ULONG Width  = (m_nHeightMapWidth  - 1) * m_nBlendTexRatio;
    ULONG Height = (m_nHeightMapHeight - 1) * m_nBlendTexRatio;
```

The first thing we do is calculate how large the alpha map for each layer will be. This is done by multiplying the size of the terrain (in quads) by the blend texture ratio extracted from the .ini file. Next we allocate some local variables whose use will become clear as we progress through the code.

```
    char  Buffer  [1025], FileName[MAX_PATH], Section [100];
    ULONG i, j, x, z, LayerCount;
    float Angle;
    UCHAR Value;
    D3DXVECTOR2 Scale;

    HRESULT             hRet;
    D3DXIMAGE_INFO      Info;
    LPDIRECT3DSURFACE9  pSurface = NULL;
```

Before we allocate the memory for each CTerrainLayer object, we extract the number of layers from the [General] section of the .ini file.

```
    // Read in the terrain layer data
    strcpy( Section, "General" );
    LayerCount = GetPrivateProfileInt( Section, "LayerCount", 1, DefFile );
```

Next we use a CTerrain helper function called AddTerrainLayer to allocate the requested CTerrainLayer objects and add their pointers to the CTerrain::m_pLayer array (resizing if necessary). We will not show the code for this function since it is a simple array resize and object allocation function that we have seen many times before. When this function returns, our m_pLayer array will have LayerCount pointers to valid CTerrainLayer objects stored in them.

```
    // Allocate our layer data
    if ( AddTerrainLayer( LayerCount ) < 0 ) return false;
```

Now we need to loop through each layer and set its properties. The first thing the following code does is obtain a pointer to the current layer. It then builds a string that describes the section in the .ini where this layer's values are stored. This will allow us to query the file through our Windows API functions. Remember that all of layer 1's settings are in the [Layer 1] section, all of layer 2's settings are in the [Layer 2] section, and so on.

```
    // Read in the element data
    for ( i = 0; i < m_nLayerCount; i++ )
    {
        CTerrainLayer * pLayer = m_pLayer[i];

        // Build section string
        if ( i == 0 )
            strcpy( Section, "Base Layer" );
        else
            sprintf( Section, "Layer %i", i );
```

Now we extract the texture index for this layer. This index describes the base texture for the layer as an index into the CTerrain::m_pTexture array. We also set the width and height of the layer (calculated at the top of the function). In this implementation, these values will be the same for all layers.

```
        // Store layer properties
        pLayer->m_nTextureIndex = (short)GetPrivateProfileInt(Section, "TextureIndex",
                                                              0, DefFile);
        pLayer->m_nLayerWidth   = Width;
        pLayer->m_nLayerHeight  = Height;
```

If the layer has a translation vector specified in the .ini file then we will extract this value and set the third row of its texture matrix. A default translation vector of (0.0, 0.0) is used otherwise. Remember that this matrix deals with 2D texture coordinates, so the third row (elements m31 and m32) contain the translation vector.

```
        // Calculate layer texture matrix
        GetPrivateProfileString(Section, "Translation", "0.0, 0.0", Buffer, 1024, DefFile);
        sscanf( Buffer, "%g,%g", &pLayer->m_mtxTexture._31, &pLayer->m_mtxTexture._32 );
```

If there is a rotation specified, we need to extract the angle from the .ini file and set the top-left 2x2 section of the texture matrix for this layer.

```
        GetPrivateProfileString( Section, "Rotation", "0.0", Buffer, 1024, DefFile );
        sscanf( Buffer, "%g", &Angle );

        // Rotate the texture matrix
        if ( Angle != 0.0f )
        {
           Angle = D3DXToRadian( Angle );
           pLayer->m_mtxTexture._11 =  cosf(Angle); pLayer->m_mtxTexture._12 = sinf(Angle);
           pLayer->m_mtxTexture._21 = -sinf(Angle); pLayer->m_mtxTexture._22 = cosf(Angle);

        } // End if apply any rotation
```

The section may also include a scale value to control tiling frequency. Once we extract the U and V values from the .ini file (default scale is (1, 1)) we apply them to the 3x2 section of the texture matrix.

```
        // Scale values
        GetPrivateProfileString( Section, "Scale", "1.0, 1.0", Buffer, 1024, DefFile );
        sscanf( Buffer, "%g,%g", &Scale.x, &Scale.y );

        pLayer->m_mtxTexture._11 *= Scale.x;
        pLayer->m_mtxTexture._21 *= Scale.x;
        pLayer->m_mtxTexture._31 *= Scale.x;
        pLayer->m_mtxTexture._12 *= Scale.y;
        pLayer->m_mtxTexture._22 *= Scale.y;
        pLayer->m_mtxTexture._32 *= Scale.y;
```

We already know how big the alpha map for this layer will be so we will allocate memory to hold one byte for each pixel. This array will be used to store the data loaded in from the alpha map files.

```
        // Allocate our layer blend map array (these are temporary arrays)
        pLayer->m_pBlendMap = new UCHAR[ Width * Height ];
        if (!pLayer->m_pBlendMap) return false;
```

Initially we will set every pixel in the layer alpha map to 0 (totally transparent). This value will be updated as we load in data from the alpha map files. However, if the current layer we are processing is the base layer of the terrain, it will not have an alpha map file and will be set to fully opaque (255) for the time being. You are reminded that the base layer never needs to blend on top of any other layer because it is always rendered first. So if this layer is the base layer, after we set its alpha map to opaque, we skip the rest of the loop because there is nothing else that needs to be done for the time being.

```
        // Set the blend map data to full transparency for now
        memset( pLayer->m_pBlendMap, 0, Width * Height );

        // Base layer is always fully opaque
        if  ( i == 0 ) { memset( pLayer->m_pBlendMap, 255, Width * Height ); continue; }
```

For every other layer except the base layer, we must load in its associated alpha map. We extract the file name from the .ini file for the current layer as shown below.

```
        // Get layer filename for non base layers
        GetPrivateProfileString(Section, "LayerMap", "", FileName, MAX_PATH - 1, DefFile);

        // Build the layer map path / filename
        strcpy( Buffer, DataPath );
        strcat( Buffer, FileName );
```

We do not want this alpha map to be a texture since it will be used later to build smaller sub textures for each terrain block. We can use the D3DX library functions to load the file into a surface format that we specify. Once the image data is loaded into that surface format, we can lock it and extract the information for each pixel and copy it into our byte array (the alpha map). The first thing we do is call D3DXGetImageInfoFromFile and pass it the name of the file and the address of a D3DXIMAGE_INFO structure. This function does not load the file; it simply opens the file and gathers information about its properties. The information is returned in the passed D3DXIMAGE_INFO structure (see Chapter Six).

```
        // Get the source file info
        if ( FAILED(D3DXGetImageInfoFromFile( Buffer, &Info ) )) return false;
```

The only information returned in the D3DXIMAGE_INFO structure that we are interested in is the width and height of the image. With these we can create a surface of the correct size to load the image into. We create an offscreen plain surface (see Chapter Six) since it does not need to be used as a texture or a render target and we are freed from device restrictions like maximum texture size.

```
        // Create the off screen surface in sys mem, in a format useful to us
        hRet = m_pD3DDevice->CreateOffscreenPlainSurface( Info.Width, Info.Height,
                                                          D3DFMT_X8R8G8B8,
                                                          D3DPOOL_SYSTEMMEM, &pSurface,
                                                          NULL  );
        if ( FAILED(hRet) ) return false;
```

We pass in the width and height of the surface we require and the common X8R8G8B8 32-bit pixel format that we would like the surface to use. Note that we specify that we would like this to be in system memory because we will be locking and reading back the pixel data. We also pass a pointer to an IDirect3DSurface9 interface which will point to a valid surface in the requested format and of the requested size.

Our next task is to use D3DXLoadSurfaceFromFile to load the file into our newly created surface. Once the function returns, regardless of the format of the alpha map file, we will have it in 32-bit X8R8G8B8 format on our temporary surface.

```
        // Load in the image
        hRet = D3DXLoadSurfaceFromFile(pSurface, NULL, NULL, Buffer,
                                       NULL, D3DX_DEFAULT, 0, NULL);
        if ( FAILED(hRet) ) { pSurface->Release(); return false; }
```

In our application, we are storing the alpha map for each layer as an array of BYTE values (one for each pixel). The surface we have just loaded our alpha map into however is a 32-bit surface. Because every pixel in that map is a shade of gray, every component of a given pixel will be the same (ex. (20, 20, 20)). Therefore, we only need one of the components to represent the alpha level of that pixel. A pixel of (20, 20, 20) will describe a transparency level of 20. Therefore, we can lock our surface, loop through each pixel, and extract one of the color components into our BYTE array alpha map. We extract the blue pixel in our code, but you are free to use any component you wish provided they are all the same.

```
        // Lock the surface and copy over the data into our blend map array
        D3DLOCKED_RECT LockedRect;
        hRet = pSurface->LockRect( &LockedRect, NULL, D3DLOCK_READONLY );
        if ( FAILED(hRet) ) { pSurface->Release(); return false; }

        ULONG * pBits = (ULONG*)LockedRect.pBits;

        // Loop through each row
        for ( z = 0; z < Info.Height; ++z )
        {
            // Loop through each column and extract just the blue pixel data
            for(x = 0; x < Info.Width; ++x )
                pLayer->m_pBlendMap[x+z*Width] = (UCHAR)(pBits[x] & (0x000000FF));

            // Move to the next row
            pBits += LockedRect.Pitch / 4;
        } // Next row
```

We unlock and release the surface after we have finished. We no longer need this 32-bit surface since we have the layer alpha map stored in a BYTE array pointed at by the CTerrainLayer::m_pBlendMap pointer.

```
        // Unlock & release the surface, we have decoded it now
        pSurface->UnlockRect();
        pSurface->Release();
```

Any value that is less than 15 is clamped to 0 while values of 220 or above are set to 255. This is an optimization step. If a quad for a given layer has an alpha level of 15, it would be virtually invisible but we would still need to render it. Since the quad is not likely to contribute much to our final image we can eliminate it. At the max range we have a similar concept. Alpha values greater than 220 generally result in the underlying layers being totally occluded. By setting it to 255 our layer occlusion testing code will catch this and eliminate all quads in the lower layers at that position (reducing our polygon count for the lower levels).

```
        // Clamp values to min and max
        for ( j = 0; j < (Width * Height); j++ )
        {
            UCHAR MinAlpha = 15;
            UCHAR MaxAlpha = 220;

            // Clamp layer value
            Value = pLayer->m_pBlendMap[ j ];
            if ( Value < MinAlpha ) Value = 0;
            if ( Value > MaxAlpha ) Value = 255;
            pLayer->m_pBlendMap[ j ] = Value;

        } // Next Alpha Value
    } // Next Layer
```

At this point, every layer has had its texture matrix and its texture index set. It also has a pointer to a byte array containing its alpha map. The base layer's blend map is currently fully opaque whilst the layers above have alpha maps that reflect the alpha image files.

Our next task is to calculate if any pixels in our layer alpha maps are occluded by totally opaque pixels in higher level layers. We loop through each layer starting at the base level and acquire a pointer to the current blend map.

```
    // Now we need to parse the layers and determine which alpha pixels are occluded
    for ( i = 0; i < m_nLayerCount; i++ )
    {
        CTerrainLayer * pLayer = m_pLayer[i];
```

Next we loop through each row of the blend map and each pixel in that row and extract the value of the pixel in the blend map.

```
        for ( z = 0; z < Height; z++ )
        {
            for ( x = 0; x < Width; x++ )
            {
                // Determine if we need to test occlusion
                Value = pLayer->m_pBlendMap[ x + z * Width ];
```

If the pixel is not fully opaque then we will test whether any of the other layers above this layer have a fully opaque pixel in the same position. If so, then we set the current pixel to 0 because it will be overdrawn by quads in a higher layer. When we test the pixels in the above layers for opacity, we extract the pixel using a filter (CTerrain::GetFilteredAlpha). GetFilteredAlpha samples neighboring

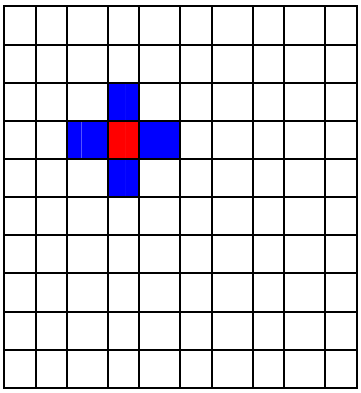
pixels in the higher level blend maps so that a pixel in a lower level layer only gets set to 0 if all the neighbouring pixels in a higher level are opaque.

```
                if ( Value > 0 )
                {
                    // Is this obscured by any layers above ?
                    Value = 0;
                    for ( j = i + 1; j < m_nLayerCount; j++ )
                    {
                        Value = m_pLayer[j]->GetFilteredAlpha( x, z );
                        if (Value == 255) break;
                    }

                    // Layer is obscured if a layer above is opaque
                    if (Value == 255) pLayer->m_pBlendMap[ x + z * Width ] = 0;

                } // End if Test Occlusion
            } // Next Column
        } // Next Row
    } // Next Layer

    // Success!!
    return true;
}
```

At this point in the code, our CTerrainLayer classes have all of their members correctly set and each layer's blend map (including the base map) will have had some of its pixels set to zero because of opaque pixels in higher layers. This means that even the base layer blend map is no longer a solid opaque block. It only contains information for building quads that are not occluded.

CTerrainLayer::GetFilteredAlpha

GetFilteredAlpha samples four neighboring pixels (plus the filter pixel) in a layer's alpha map and returns the average. In the following image, the red pixel (4, 4) is the current pixel we are filtering.



The code simply sums the five pixels and returns the averaged result.

```
UCHAR CTerrainLayer::GetFilteredAlpha( ULONG x, ULONG z )
{
    long Total, Sum, PosX, PosZ;

    // Validate Parameters
    if ( !m_pBlendMap ) return 0;

    // Loop through each neighbor
    PosX = x; PosZ = z;
    Total = m_pBlendMap[ PosX + PosZ * m_nLayerWidth ];
    Sum = 1;

    // Above Pixel
    PosX = x; PosZ = z - 1;
    if ( PosZ >= 0 )
    {
        Total += m_pBlendMap[ PosX + PosZ * m_nLayerWidth ];
        Sum++;

    } // End if Not OOB

    // Right Pixel
    PosX = x + 1; PosZ = z;
    if ( PosX < (signed)m_nLayerWidth )
    {
        Total += m_pBlendMap[ PosX + PosZ * m_nLayerWidth ];
        Sum++;

    } // End if Not OOB

    // Bottom Pixel
    PosX = x; PosZ = z + 1;
    if ( PosZ < (signed)m_nLayerHeight )
    {
        Total += m_pBlendMap[ PosX + PosZ * m_nLayerWidth ];
        Sum++;

    } // End if Not OOB

    // Left Pixel
    PosX = x - 1; PosZ = z;
    if ( PosX >= 0 )
    {
        Total += m_pBlendMap[ PosX + PosZ * m_nLayerWidth ];
        Sum++;
    } // End if Not OOB

    // Return result
    return (UCHAR)(Total / Sum);
}
```

CTerrain::GenerateTerrainBlocks

Once the CTerrain::GenerateLayers call returns back to CTerrain::LoadTerrain, the next function called is CTerrain::GenerateTerrainBlocks. This function will create the individual terrain blocks and their vertex buffers using the height map. It will also generate the splat levels for each terrain block.

The first thing we do is locally record the block width and height that the terrain will be divided into.

```
bool CTerrain::GenerateTerrainBlocks( )
{
    ULONG x, z, ax, az, Counter;

    // Calculate block values
    m_nBlocksWide = (USHORT)(m_nHeightMapWidth - 1) / m_nQuadsWide;
    m_nBlocksHigh = (USHORT)(m_nHeightMapHeight - 1) / m_nQuadsHigh;
```

In our example we will use 8x8 = 64 terrain blocks. Our next step is to allocate the 64 terrain blocks and add their pointers to the CTerrain::m_pBlock array (resizing if necessary).

```
    // Allocate enough blocks to store the separate parts of this terrain
    if ( AddTerrainBlock(  m_nBlocksWide * m_nBlocksHigh )  < 0 ) return false;
```

Once we have our array of terrain block pointers, we loop through each block, row-by-row and column-by- column and will acquire a pointer to the relevant terrain block.

```
    // Initialize each terrain block
    for ( z = 0; z < m_nBlocksHigh; z++ )
    {
        for ( x = 0; x < m_nBlocksHigh; x++ )
        {
            CTerrainBlock * pBlock = m_pBlock[ x + z * m_nBlocksWide ];
```

Now we will fill out the terrain blocks neighbor array.

```
            // Calculate Neighbour Information
            Counter = 0;
            for ( az = -1; az <= 1; az++ )
            {
                for ( ax = -1; ax <= 1; ax++, Counter++ )
                {
                    // Reset to NULL to begin with
                    pBlock->m_pNeighbours[Counter] = NULL;

                    // Bail if we are out of bounds
                    if((x+ax) < 0 || (z+az) < 0 ||
                       (x+ax) >= m_nBlocksWide || (z+az) >= m_nBlocksHigh)
                        continue;

                    // Store Neighbour
                    pBlock->m_pNeighbours[Counter] = m_pBlock[(x+ax)+(z+az)*m_nBlocksWide];

                } // Next Adjacent Column
```

```
            } // Next Adjacent Row

        } // Next Column

    } // Next Row
```

Once we have the neighbor array filled out for the current terrain block, we loop through each block again and call CTerrainBlock::GenerateBlock. This builds the vertex buffer and the child spat objects.

```
    // Generate each terrain block
    for ( z = 0; z < m_nBlocksHigh; z++ )
    {
        for ( x = 0; x < m_nBlocksHigh; x++ )
        {
            CTerrainBlock * pBlock = m_pBlock[ x + z * m_nBlocksWide ];

            // Generate the block
            if (!pBlock->GenerateBlock( this, x * m_nQuadsWide, z * m_nQuadsHigh,
                                        m_nBlockWidth, m_nBlockHeight ))
                return false;
        } // Next Column

    } // Next Row

    // Success!!
    return true;
}
```

When we call CTerrainBlock::GenerateBlock, we pass a pointer to the terrain class and the offset into the height map for the top-left vertex in the current terrain block. We also pass in the block width and height. We calculate which pixel in the height map will map to the top-left vertex of the terrain block by multiplying the offset of the current block in the terrain (x, z) by the number of quads in each terrain block (16x16). If we are processing terrain block (2,3) for example, the section of the height map that will be used to build will be (2*16) , (3*16) = pixel offset (32, 64) in the height map. We also pass the vertex width and height so that the function will know how many vertices to allocate for the vertex buffer.

CTerrainBlock::GenerateBlock

This function builds the vertex buffer and splat meshes for a given terrain block. This implementation uses precalculated vertex lighting but it could be quickly modified to use vertex normals and the lighting pipeline if preferred.

```
bool CTerrainBlock::GenerateBlock( CTerrain * pParent, ULONG StartX, ULONG StartZ,
                                   ULONG BlockWidth,  ULONG BlockHeight )
{
    ULONG             x, z;
    HRESULT           hRet;
    ULONG             Usage      = D3DUSAGE_WRITEONLY;
```

```
    USHORT           *pIndex     = NULL;
    CVertex          *pVertex    = NULL;
    float            *pHeightMap = NULL;
    LPDIRECT3DDEVICE9 pD3DDevice = NULL;
    D3DXVECTOR3       VertexPos, LightDir = D3DXVECTOR3(0.650945f, -0.390567f, 0.650945f);
```

We set up a light direction vector to calculate vertex colors using a lighting technique analogous to the DirectX directional light type. Next we check that this terrain block has a parent terrain and that the parent has a height map. If not, we exit.

```
    // Validate requirements
    if (!pParent || !pParent->GetD3DDevice() || !pParent->GetHeightMap()) return false;
```

We keep track of some local housekeeping variables that will be used during block creation.

```
    // Store some values
    m_pParent      = pParent;
    m_nStartX      = StartX;
    m_nStartZ      = StartZ;
    m_nBlockWidth  = BlockWidth;
    m_nBlockHeight = BlockHeight;
    m_nQuadsHigh   = BlockHeight - 1;
    m_nQuadsWide   = BlockWidth - 1;
    m_nQuadsHigh   = BlockHeight - 1;
    pHeightMap     = pParent->GetHeightMap();
```

We retrieve the parent's Direct3D device and query the hardware status so that we can correctly create our vertex buffers. We create the vertex buffer (17x17 vertices in this example) in the managed pool. Its interface pointer is assigned to the terrain block m_pVertexBuffer pointer.

```
    pD3DDevice     = pParent->GetD3DDevice();

    // Calculate buffer usage
    if ( !m_pParent->UseHardwareTnL() ) Usage |= D3DUSAGE_SOFTWAREPROCESSING;

    // Create the vertex buffer ready for generation
    hRet = pD3DDevice->CreateVertexBuffer((BlockWidth * BlockHeight) * sizeof(CVertex),
                                          Usage, VERTEX_FVF, D3DPOOL_MANAGED,
                                          &m_pVertexBuffer, NULL );
    if (FAILED(hRet)) return false;
```

We lock the vertex buffer to begin filling in our data. Each pixel in the block of the parent terrain's height map that maps to this terrain block will be assigned to the corresponding vertex in the vertex buffer.

```
    // Lock the vertex buffer ready to fill data
    hRet = m_pVertexBuffer->Lock(0, (BlockWidth*BlockHeight)*sizeof(CVertex),
                                 (LPVOID*)&pVertex, 0);
    if (FAILED(hRet)) return false;

    // Loop through and generate the vertex data
    for ( z = StartZ; z < StartZ + BlockHeight; z++ )
```

```
    {
        for ( x = StartX; x < StartX + BlockWidth; x++ )
        {
```

To calculate the position of the vertex in the world, we take the current X and Y pixel offsets in the height map and copy them into the X and Z components of the vertex. These values are scaled by the X and Z components of our scale vector. We read the height value from the pixel in the height map and multiply it with the Y component of our scale vector to produce the world space height of the vertex. The vertex color is calculated and clamped exactly as it was in Chapter Three. All of this information is then copied to the vertex buffer.

```
            VertexPos.x = (float)x * m_pParent->GetScale().x;
            VertexPos.y = pHeightMap[x+z*pParent->GetTerrainWidth()]*m_pParent->GetScale().y;
            VertexPos.z = (float)z * m_pParent->GetScale().z;

            // Calculate vertex color scale
            float fRed = 1.0f, fGreen = 1.0f, fBlue = 1.0f, fScale = 0.25f;

            // Generate average scale (for diffuse lighting calc)
            fScale  = D3DXVec3Dot(&pParent->GetHeightMapNormal( x, z ), &(-LightDir));
            fScale += D3DXVec3Dot(&pParent->GetHeightMapNormal( x + 1, z ), &(-LightDir));
            fScale += D3DXVec3Dot(&pParent->GetHeightMapNormal(x + 1, z + 1 ), &(-LightDir));
            fScale += D3DXVec3Dot( &pParent->GetHeightMapNormal( x, z + 1 ), &(-LightDir));
            fScale /= 4.0f;

            // Increase Saturation
            fScale += 0.25f; //0.05f;

            // Clamp color saturation
            if ( fScale > 1.0f ) fScale = 1.0f;
            if ( fScale < 0.4f ) fScale = 0.4f;

            // Store Vertex Values
            pVertex->x       = VertexPos.x;
            pVertex->y       = VertexPos.y;
            pVertex->z       = VertexPos.z;
            pVertex->Diffuse = D3DCOLOR_COLORVALUE(fRed*fScale, fGreen*fScale,
                                                   fBlue*fScale, 1.0f);
```

All that is left to do for the current vertex is calculate the two sets of texture coordinates for the base texture and the alpha map. By default we will set the UV coordinates of the first texture coordinate to the coordinates of the pixel in the height map for which the vertex was generated. As this first set of texture coordinates is used in texture stage 0 to tile the base texture across the terrain, each quad will have a whole texture mapped to it. Note that this is only the base mapping; the texture matrix for each layer will (possibly) be used to transform this one texture per quad relationship into something else. In our application we scale the UV coordinate by 0.5 on both U and V so that a single quad has a texture tiled twice both horizontally and vertically. So there will be a 4 quad per texture tiling ratio for our layers. We can change this tiling for each layer by altering the ini file.

```
            pVertex->tu      = (float)x;
            pVertex->tv      = (float)z;
```

Calculating the second set of texture coordinates is also straightforward but perhaps not obvious at first. We know that our layer maps will be divided into terrain block sized blend map textures. For a given terrain block, a layer's blend map will be draped across it such that the blend map texture coordinates should map from 0.0 to 1.0 for the second texture coordinate set in the terrain block. In other words, each terrain block will have three blend textures, and each blend texture will be mapped to the four corners of the terrain block. Thus all we need to do is subtract the offset of the current block in the height map from the current loop counter such that we have coordinates in the range of [0, 16]. We then divide the result by 16 to produce coordinates in the range [0.0, 1.0].

```
            pVertex->tu2     = (float)(x - StartX) / m_nQuadsWide;
            pVertex->tv2     = (float)(z - StartZ) / m_nQuadsHigh;

            // Move to next vertex
            pVertex++;
        } // Next Column

    } // Next Row

    // Finished with the vertex buffer
    m_pVertexBuffer->Unlock();
```

When the loop exits, we have a full vertex buffer and we can unlock it. Before this function returns control back to CTerrain::GenerateLayers, we must call three housekeeping functions. CTerrainBlock::CountLayerUsage checks each layer map pixel mapped to the terrain block and records the number of non-transparent pixels. This call fills out the CTerrainBlock::m_pLayerUsage array so that when we are rendering a given layer, we can check the appropriate value in this array. If it is 0, there is nothing in this block to render for the current layer.

```
    // Determine all the layers used by this block
    if ( !CountLayerUsage() ) return false;
```

The next function we will call is CTerrainBlock::GenerateSplats to build the index buffers for each layer in the terrain block. It will build only the renderable quads for a given layer.

```
    // Generate Splat Levels for this block
    if ( !GenerateSplats() ) return false;
```

At this point we have a list of child splats for this terrain block. Each contains an index buffer describing the quads for that splat. Our final task is generating the blend textures for each splat. This is a simple enough case of copying a section of the CTerrainLayer blend maps into a blend texture containing only the pixel alpha information. If we have three layers, each terrain block will have three CTerrainSplat objects. Each splat object will contain an index buffer and a blend texture describing the alpha information that maps to that terrain block.

```
    // Generate the blend maps
    if ( !GenerateBlendMaps() ) return false;

    return true;
}
```

CTerrainBlock::CountLayerUsage

CountLayerUsage fills out the CTerrainBlock::m_pLayerUsage array with a value describing the number of non-transparent pixels in a given layer that map to this terrain block. If any of the values in the CTerrainBlock::m_pLayerUsage are still zero when the function completes, then this terrain block does not have any quads to render for the corresponding layer. This allows us to know whether a terrain block needs to have its Render function called for a given layer.

```
bool CTerrainBlock::CountLayerUsage()
{
    USHORT i;
    ULONG  x, z;
    UCHAR  Value;
```

First we set the number of layers this terrain uses and allocate the terrain block m_pLayerUsage array to hold that many unsigned shorts. We initially set all values in this array to zero.

```
    // Allocate the layer usage array
    m_pLayerUsage = new USHORT[ m_pParent->GetLayerCount() ];
    if( !m_pLayerUsage ) return false;
    ZeroMemory( m_pLayerUsage, m_pParent->GetLayerCount() * sizeof(USHORT));
```

Our next task is to calculate the rectangle in our 1024x1024 layer alpha maps that will map to this terrain block. StartX and StartY already contain the offsets into the height of the rectangle used to build this terrain block. Therefore all we have to do is multiply these values by the blend texture ratio (8 in our application) and we have the rectangle in each layer map that will be mapped to this terrain block.

```
    // Pre-Calculate loop counts
    ULONG LoopStartX = (m_nStartX * m_pParent->GetBlendTexRatio());
    ULONG LoopStartZ = (m_nStartZ * m_pParent->GetBlendTexRatio());
    ULONG LoopEndX   = (m_nStartX + m_nQuadsWide) * m_pParent->GetBlendTexRatio();
    ULONG LoopEndZ   = (m_nStartZ + m_nQuadsHigh) * m_pParent->GetBlendTexRatio();
```

Now we will loop through each pixel in the rectangle and, for each pixel offset, check the value of the pixel in each of the layer alpha maps. If the value is not zero, then we increment the value for that layer in the m_pLayerUsage array. At the end of the function, we will have three values for each of our terrain blocks. These describe how many pixels in each layer's alpha map are non-zero.

```
    // Determine which layers we are using in this block
    for ( z = LoopStartZ; z < LoopEndZ; z++ )
    {
        for ( x = LoopStartX; x < LoopEndX; x++ )
        {
            // Loop through each layer
            for ( i = 0; i < m_pParent->GetLayerCount(); i++ )
            {
                CTerrainLayer * pLayer = m_pParent->GetLayer(i);

                // Retrieve alpha value
```

```
                Value = pLayer->m_pBlendMap[ x + z * pLayer->m_nLayerWidth ];
                if ( Value > 0 ) m_pLayerUsage[i]++;
            } // Next Layer
        } // Next Column
    } // Next Row

    // Success!!
    return true;
}
```

CTerrainBlock::GenerateSplats

CTerrainBlock::GenerateSplats creates the child splat objects for each terrain block. This function is very small because most of the work happens in CTerrainBlock::GenerateSplatLevel (which builds the index buffer for each splat).

The function retrieves the number of layers used by the terrain and calls CTerrainBlock::AddSplatLevel to allocate an array large enough to hold a pointer for each CTerrainSplat object. The code to the CTerrainBlock::AddSplatLevel function will not be covered here. It is a basic housekeeping function that allocates an array and we have looked at enough of those at this point to give you a good idea of how it works.

```
bool CTerrainBlock::GenerateSplats()
{
    USHORT i;

    // Allocate the required number of splat levels
    if ( AddSplatLevel( m_pParent->GetLayerCount() ) < 0 ) return false;
```

Next we loop through each layer and check the m_pLayerUsage array to see if the current layer is used by the terrain block. If there is a non-zero usage value, then we call the CTerrainBlock::GenerateSplatLevel function to allocate a CTerrainSplat object for that layer and generate its index buffer. A zero value indicates that this terrain block does not have any quads for this layer and we continue looping.

```
    for ( i = 0; i < m_pParent->GetLayerCount(); i++ ){
        // Is this layer in use ?
        if ( !m_pLayerUsage[i] ) continue;

        // Generate the splat level for this layer
        if (!GenerateSplatLevel( i )) return false;
    }

    return true;
}
```

When this function returns, the current terrain block will have three child splat objects (in our application). Each splat object will have an index buffer (if applicable) describing a series of quads that need to be rendered for the corresponding layer.

CTerrainBlock::GenerateSplatLevel

This function is responsible for creating and filling the index buffer with quads for the specified splat level. The function is passed the layer index that this splat will be assigned to.

```
bool CTerrainBlock::GenerateSplatLevel( USHORT TerrainLayer )
{
    HRESULT   hRet;
    USHORT   *pIndex = NULL;
    ULONG     x, z, ax, az;
    UCHAR     Value;
    float     BlendTexels = m_pParent->GetBlendTexRatio();

    LPDIRECT3DDEVICE9 pD3DDevice = m_pParent->GetD3DDevice();
    bool HardwareTnL = m_pParent->UseHardwareTnL();
    CTerrainLayer * pLayer = m_pParent->GetLayer( TerrainLayer );

    // Calculate usage variable
    ULONG Usage = D3DUSAGE_WRITEONLY;
    if (!HardwareTnL) Usage |= D3DUSAGE_SOFTWAREPROCESSING;
```

We record whether we are using a hardware or software vertex processing device as we will need this information when we generate the index buffer. We begin by allocating a new CTerrainSplat object and assigning its pointer to the terrain block m_pSplatLevel array at the index that corresponds to the layer for which it is being built.

```
    // Allocate a new splat
    CTerrainSplat * pSplat = new CTerrainSplat;
    if (!pSplat) return false;

    // Store the splat objects pointer the CTerrainBlock::m_pSplatArray
    m_pSplatLevel[ TerrainLayer ] = pSplat;

    // Store layer index (handy later on)
    pSplat->m_nLayerIndex = TerrainLayer;
```

The index buffer will store 6 indices for every quad because we will be rendering our splat quads as indexed triangle lists. We can no longer render connected primitives (such as a strip) because only a handful of quads may be built for a given layer and they may be scattered about the block.

```
    // Create the index buffer ready for generation
    hRet = pD3DDevice->CreateIndexBuffer(((m_nQuadsWide*m_nQuadsHigh)*6) * sizeof(USHORT),
                                         Usage, D3DFMT_INDEX16, D3DPOOL_MANAGED,
                                         &pSplat->m_pIndexBuffer, NULL );
    if ( FAILED(hRet ) ) return false;

    // Lock the index buffer ready to fill data
    hRet = pSplat->m_pIndexBuffer->Lock(0,((m_nQuadsWide*m_nQuadsHigh)*6) * sizeof(USHORT),
                                        (void**)&pIndex, 0 );
    if ( FAILED(hRet ) ) return false;

    // Calculate the indices for the splat tri-list
    for ( z = 0; z < m_nQuadsHigh; z++ )
```

```
    {
        // Pre-Calc Loop starts / ends
        ULONG LoopStartZ = ( z + m_nStartZ ) * BlendTexels;
        ULONG LoopEndZ   = LoopStartZ + BlendTexels;

        for ( x = 0; x < m_nQuadsWide; x++ )
        {
            // Pre-Calc Loop starts / ends
            ULONG LoopStartX = ( x + m_nStartX ) * BlendTexels;
            ULONG LoopEndX   = LoopStartX + BlendTexels;
```

We are calculating the rectangle in this layer's alpha map that maps to the current quad for this terrain block. In our demo a single quad has an 8x8 rectangle in the alpha map that maps to the quad. m_nStartZ and m_nStartX contain the top left corner offset into the height map for the terrain block. X and Z are the loop variables that allow us to step through each quad starting from this offset. Since the layer maps are larger than the height map, we multiply by the BlendTexels variable (8 in our demo) so that LoopStartZ and LoopStartX describe the top left corner of the rectangle in the image map. LoopEndX and LoopEndZ describe the bottom right pixel of the rectangle in the layer alpha map. All we do now is test each of the 8x8 texels in the layer alpha map. If we find that all of these texels are zero, then there is no need to build a quad there (it would be totally transparent).

```
            // Determine if element is visible anywhere
            for ( az = LoopStartZ; az < LoopEndZ; az++ )
            {
                for ( ax = LoopStartX; ax < LoopEndX; ax++ )
                {
                    // Retrieve the layer data
                    Value = pLayer->m_pBlendMap[ ax + az * pLayer->m_nLayerWidth ];
                    if ( Value > 0 ) break;
                } // Next Alpha Column

                // Break if we found one
                if ( Value > 0 ) break;
            } // Next Alpha Row

            // Should we write the quad here ?
            if ( Value == 0 ) continue;
```

If we get here, then at least one of the 8x8 texels in the blend map region that maps to this quad is not totally transparent. Therefore, we need to add the 6 indices for the quad to the splat index buffer. We also increase the CTerrainSplat::m_nIndexCount and CTerrainSplat::m_nPrimitiveCount members so that they accurately reflect the number of indices/primitives in the index buffer.

```
            // Insert next two triangles
            *pIndex++ = (USHORT)(x + z * m_nBlockWidth);
            *pIndex++ = (USHORT)(x + (z + 1) * m_nBlockWidth);
            *pIndex++ = (USHORT)((x + 1) + (z + 1) * m_nBlockWidth);

            *pIndex++ = (USHORT)(x + z * m_nBlockWidth);
            *pIndex++ = (USHORT)((x + 1) + (z + 1) * m_nBlockWidth);
            *pIndex++ = (USHORT)((x + 1) + z * m_nBlockWidth);

            // Increase our index & Primitive counts
```

```
            pSplat->m_nIndexCount     += 6;
            pSplat->m_nPrimitiveCount += 2;

        } // Next Element Column
    } // Next Element Row
```

Finally, we unlock the buffer and we are done.

```
    // Unlock the index buffer
    pSplat->m_pIndexBuffer->Unlock();

    // Success!!
    return true;
}
```

CTerrainBlock::GenerateBlendMaps

This function generates the blend maps for each terrain block -- one for each layer. This is where the global blend maps are carved up into terrain block sized maps to be stored in DirectX texture objects for rendering.

```
bool CTerrainBlock::GenerateBlendMaps( )
{
    HRESULT hRet;
    ULONG Width, Height, i, x, z;
    LPDIRECT3DDEVICE9 pD3DDevice = m_pParent->GetD3DDevice();
    D3DLOCKED_RECT LockData;
    UCHAR Value;
    ULONG BlendTexels = m_pParent->GetBlendTexRatio();

    // Bail if we have no data
    if ( m_nSplatCount == 0 ) return true;
```

We will need to create textures for each layer to hold the blend maps for this terrain block, so we must first calculate how large these textures need to be. We know how many quads wide and high our terrain blocks are and we also know that (in our demo) we have an alpha ratio of 8. This means that an 8x8 block of alpha texels map to a single quad. Therefore, we can calculate the texture size we need as follows.

```
    // Calculate width / height of the texture
    Width  = (m_nQuadsWide * BlendTexels);
    Height = (m_nQuadsHigh * BlendTexels);
```

We now loop through each element in the terrain block splat level array and skip to the next layer if the current splat level is NULL. In that case, the terrain block has no quad data for the corresponding layer.

```
    // Calculate each splats blend map
    for ( i = 0; i < m_nSplatCount; i++ )
```

```
    {
        // Bail if this is an empty splat level
        if ( !m_pSplatLevel[i] ) continue;
```

Next we get a pointer to the current layer we are processing. We skip to the next iteration of the loop if this is the base layer because the base layer will not have a blend map.

```
        CTerrainLayer * pLayer = m_pParent->GetLayer( i );

        // We never generate an alpha map for terrain layer 0
        if ( m_pSplatLevel[i]->m_nLayerIndex == 0) continue;
```

If we get here, then this splat level has an index buffer and it is not the splat level associated with the base layer of the terrain block. Therefore, we will need to create a texture to hold the blend map data. In this example we create a D3DFMT_A4R4G4B4 texture. We will be configuring the second texture stage to sample the alpha values from this texture and the RGB information will not be used at all. Again, we could use a pure alpha surface in the format D3DFMT_A8 and this would certainly be more memory efficient. The only reason we have not done so in this demo is that pure alpha surface support is spotty amongst current graphics cards and drivers. So for compatibility we use a 16-bit ARGB texture to keep the size down. When pure alpha surfaces become more widely supported, it would be a simple task to replace the 16-bit texture.

```
        // Create our blend texture
        hRet = pD3DDevice->CreateTexture(Width, Height, 1, 0, D3DFMT_A4R4G4B4,
                                         D3DPOOL_MANAGED,&m_pSplatLevel[i]->m_pBlendTexture,
                                         NULL);
        if ( FAILED(hRet) ) return false;
```

The returned texture interface pointer is stored in the current splat object m_pBlendTexture member. Our next task is to lock the texture and copy the relevant data from our large layer alpha map into the blend texture. The alpha values in our large alpha maps are currently byte values so we will need to quantize them into 4-bit values so that they fit in the top 4 bits of the 16-bit pixel.

```
        // Lock the texture
        hRet = m_pSplatLevel[i]->m_pBlendTexture->LockRect( 0, &LockData, NULL, 0 );
        if ( FAILED(hRet) ) return false;

        USHORT * pBuffer = (USHORT*)LockData.pBits;
```

Now that we have a pointer to the texture bits, we loop through each pixel in each row and extract the byte from the large layer alpha map. Notice that we are only extracting values from the large layer map within the rectangle that maps to this terrain block.

```
        // Loop through each pixel and store
        for ( z = 0; z < Height; z++ )
        {
            for ( x = 0; x < Width; x++, pBuffer++ )
            {
                // Retrieve alpha value
                Value = pLayer->m_pBlendMap[(x + (m_nStartX * BlendTexels)) + \
```

```
                                        (z + (m_nStartZ * BlendTexels))*pLayer->m_nLayerWidth];
```

We convert from an 8-bit value to a 4-bit value by shifting it to the left by 8-bits so that it will be contained in the top byte of the two byte pixel. We then mask off the bottom 12 bits. We are left with a WORD value where the red, green, and blue components are set to zero and the 4 alpha bits contain a scaled down alpha value in the range of 0 to 15.

```
                // Store value in buffer ( Shift right 4 and left 12 )
                *pBuffer = ((LONG)Value << 8) & 0xF000;
            } // Next Column
```

We advance the pointer to the texture bits in the next row by taking pitch into account.

```
            // Add on pitch tail
            pBuffer += LockData.Pitch - ( Width * sizeof(USHORT) );
        } // Next Row
```

At this point we are done with the splat level for the current layer so we unlock the texture surface. We then repeat the procedure for the remaining layers (splat levels).

```
        // Unlock the blend texture
        m_pSplatLevel[i]->m_pBlendTexture->UnlockRect( 0 );

    } // Next Splat Level

    // Success!!
    return true;
}
```

CTerrain::Render

The CGameApp::FrameAdvance function renders the terrain with a call to the CTerrain::Render function. It accepts a single parameter which is a pointer to our CCamera object.

The first thing the function does is enable alpha blending and setup the alpha blending equation to use the common D3DBLEND_SRCALPHA and D3DBLEND_INVSRCALPHA blend modes for the source and destination blend modes respectively.

```
void CTerrain::Render( CCamera * pCamera )
{
    USHORT i;
    ULONG  j;

    // Validate parameters
    if( !m_pD3DDevice ) return;

    // Setup our terrain render states
    m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, true );
    m_pD3DDevice->SetRenderState( D3DRS_SRCBLEND, D3DBLEND_SRCALPHA );
```

```
    m_pD3DDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
```

The texture stages are setup such that fragment color is sampled from the base texture set in texture stage 0 and alpha is sampled from the blend texture set in texture stage 1. The color in the first stage is modulated with the interpolated diffuse vertex color for lighting purposes.

```
    // stage 0 coloring : get color from texture0*diffuse
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
```

We will not need to sample alpha from texture stage 0. However we must be careful not to disable the alpha operations for a stage since this will cut off the alpha operations in higher texture stages. So we will set the alpha operation for stage 0 to D3DTA_CURRENT (the equivalent of D3DTA_DIFFUSE) and the interpolated alpha value of the vertex will be used. As our vertices have fully opaque diffuse colors, this will equate to an alpha value of 255 being passed to the second stage. This value is ignored and we will sample the alpha from the texture stored there instead.

```
    // stage 0 alpha : nada
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1 );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_CURRENT );
```

In the second stage we simply select the color from the first stage and output it unaltered.

```
    // stage 1 coloring : nada
    m_pD3DDevice->SetTextureStageState( 1, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
    m_pD3DDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_CURRENT );
```

The alpha operations in texture stage 1 sample the alpha value from the texture assigned to that stage. This of course is our blend texture.

```
    // stage 1 alpha : get alpha from texture1
    m_pD3DDevice->SetTextureStageState( 1, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1 );
    m_pD3DDevice->SetTextureStageState( 1, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );
```

Our next task is to setup texture stage 0 (recall that it holds the base texture) to handle texture transformations. Each layer has a texture matrix that will be used to transform the first set of UV coordinates to control base texture tiling. Therefore, we set the **D3DTSS_TEXTURETRANSFORMFLAGS** texture stage state to **D3DTFF_COUNT2** to inform the pipeline that we will be requiring our first set of texture coordinates to be multiplied by the texture matrix for stage 0 and that we are using 2D coordinates. We do not enable texture transforms for stage 1 because the alpha blend texture in that stage is mapped to the four corners of each terrain block. This must not be changed.

```
    // Enable Stage Texture Transforms
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT2 );
```

Next we inform the pipeline of the vertex types we will be using so that it knows which components each vertex of our terrain will contain. In our case this will be a position, a diffuse color, and two sets of 2D texture coordinates.

```
    // Setup our terrain vertex FVF code
    m_pD3DDevice->SetFVF( VERTEX_FVF );
```

Finally, we loop through each terrain block in our terrain block array, assign its vertex buffer to stream 0, and then traverse each layer. If the current terrain block uses the current layer we set the layer's base texture to texture stage 0, assign the texture matrix, and call CTerrainBlock::Render to draw the block. We pass the index of the layer we are currently rendering because CTerrainBlock::Render renders an individual splat level.

```
    // Loop through blocks and signal a render
    for ( j = 0; j < 1/* m_nBlockCount*/; j++ )
    {
        m_pD3DDevice->SetStreamSource(0, m_pBlock[j]->m_pVertexBuffer, 0, sizeof(CVertex));

        // Loop through all active layers
        for ( i = 0; i < m_nLayerCount; i++ )
        {
            // Skip if this layer is disabled
            if ( GetGameApp()->GetRenderLayer( i ) == false ) continue;

            CTerrainLayer * pLayer = m_pLayer[i];
            if ( !m_pBlock[j]->m_pLayerUsage[ i ] ) continue;

            // Set our texturing information
            m_pD3DDevice->SetTexture( 0, m_pTexture[pLayer->m_nTextureIndex] );
            m_pD3DDevice->SetTransform( D3DTS_TEXTURE0, &pLayer->m_mtxTexture );
            m_pBlock[j]->Render( m_pD3DDevice, i );

        } // Next Block
    } // Next Layer
}
```

CTerrainBlock::Render

This function is called for each layer in each terrain block to render the indicated splat. The function sets the splat index buffer as the current device index buffer and assigns the splat blend texture to texture stage 1. It concludes with a call to DrawIndexedPrimitive to render the quads.

```
void CTerrainBlock::Render( LPDIRECT3DDEVICE9 pD3DDevice, USHORT LayerIndex )
{
    // Bail if this layer is not in use
    if ( !m_pSplatLevel[LayerIndex] ) return;

    // Set up vertex streams & Textures
    pD3DDevice->SetIndices( m_pSplatLevel[LayerIndex]->m_pIndexBuffer );
    pD3DDevice->SetTexture( 1, m_pSplatLevel[LayerIndex]->m_pBlendTexture );
```

```
    // Render the vertex buffer
    if ( m_pSplatLevel[LayerIndex]->m_nPrimitiveCount == 0 ) return;
    pD3DDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, 0,
                                      (m_nBlockWidth * m_nBlockHeight), 0,
                                       m_pSplatLevel[LayerIndex]->m_nPrimitiveCount );
}
```

Questions and Exercises

1. Can we use vertex and texture alpha simultaneously when performing alpha blending?
2. What does it mean if a texture format is said to have an alpha channel?
3. If a texture uses the format **X8R8G8B8**, does it contain per-pixel alpha information?
4. List four locations/sources where alpha values can be stored and retrieved by the texture blending cascade.
5. If we use alpha values stored in materials, is this alpha information described as per-vertex alpha, per- pixel or per-triangle/face?
6. How does the **D3DRS_TEXTUREFACTOR** render state allow us to make a constant alpha value available to all texture stages? How does a texture stage select this alpha value as an alpha argument?
7. What does the D3DTSS_CONSTANT texture stage state allow us to do?
8. Describe how texture stage 0 would retrieve its color and alpha information using the following texture stage states.

```
pDevice->SetTextureStageState ( 0 , D3DTSS_COLORARG1 , D3DTA_TEXTURE);
pDevice->SetTextureStageState ( 0 , D3DTSS_COLOROP   , D3DTOP_SELECTARG1);
pDevice->SetTextureStageState ( 0 , D3DTSS_ALPHAARG1 , D3DTA_DIFFUSE);
pDevice->SetTextureStageState ( 0 , D3DTSS_ALPHAOP   , D3DTA_SELECTARG1);
```

9. Describe the color and alpha values output from the texture cascade using the following render states for stage 0 and stage 1.

```
pDevice->SetRenderState( D3DRS_TEXTUREFACTOR , 0x400000FF );

pDevice->SetTextureStageState ( 0 , D3DTSS_COLORARG1 , D3DTA_TEXTURE);
pDevice->SetTextureStageState ( 0 , D3DTSS_COLORARG1 , D3DTA_DIFFUSE);
pDevice->SetTextureStageState ( 0 , D3DTSS_COLOROP   , D3DTOP_MODULATE;
pDevice->SetTextureStageState ( 1 , D3DTSS_COLORARG1 , D3DTA_CURRENT);
pDevice->SetTextureStageState ( 1 , D3DTSS_COLORARG1 , D3DTA_TFACTOR);
pDevice->SetTextureStageState ( 1 , D3DTSS_COLOROP   ,D3DTOP_ADD );

pDevice->SetTextureStageState ( 0 , D3DTSS_ALPHAARG1 , D3DTA_TEXTURE);
pDevice->SetTextureStageState ( 0 , D3DTSS_ALPHAOP   , D3DTA_SELECTARG1);
pDevice->SetTextureStageState ( 1 , D3DTSS_ALPHAARG1 , D3DTA_CURRENT);
pDevice->SetTextureStageState ( 1 , D3DTSS_ALPHAARG2 , D3DTA_TFACTOR);
pDevice->SetTextureStageState ( 1 , D3DTSS_ALPHAOP   , D3DTA_MODULATE);
```

10. Why should the following equation be familiar to us and considered significant?

SourceColor * SrcBlendMode + DestColor * DestBlendMode

11. What is alpha testing and when can it be useful?
12. When polygons are partially transparent, why do we need to render the alpha polygons in a second pass?
13. Why would we ever need to sort alpha polygons, even when rendering them in a second pass?
14. Which is the better sorting algorithm to use when many alpha polygons need to be sorted prior to rendering: a bubble sort or a quick sort?
15. What is a hash table and how does it enable us to quickly sort polygons prior to rendering?
16. Do we need to sort polygons if we are performing additive color blending?
17. What is a pure alpha surface?
18. DirectX graphics provides two fog modes, what are they?
19. Excluding the lack of any fog as a fog model, how many fog models are available for each fog mode?
20. What is the Fog Factor?
21. If you were not using the transformation pipeline but still wanted vertex fog, you could enable fog and calculate your own vertex fog factors. Where would you store these per-vertex fog factors in order for them be accessed and used for fogging by the pipeline?
22. Why is pixel fog often referred to as table fog?
23. Do we need to set a different fog color for both vertex fog mode and table fog mode or do they share the same fog color render state?
24. What are the differences between setting up the linear fog model and setting up either of the exponential fog models for a given fog mode?
25. Do you need to set a fog density value when using linear fog?
26. When using vertex fog mode and the linear fog model, we specify the fog start and fog end distances as device coordinates in the 0.0 – 1.0 range. True or False?
27. What is a W-friendly projection matrix?
28. When using vertex fog, what causes rotation artifacts and how can we potentially avoid it?
29. Regardless of whether we are using vertex fog mode or pixel fog mode, we set up all fog parameters by setting render states. True or False?

Appendix A: Texture Stage States, Render States and Sampler States

Below is a list of texture stage states, render states and sampler states introduced in this chapter.

RenderState	Parameters	Description
D3DRS_ALPHABLENDENABLE	True or False	Enable alpha blending in the pipeline. When enabled, the color and alpha values output from the texture stage cascade are used in a blending operation with the frame buffer to generate the pixel color. When disabled, the alpha output from the texture stage is discarded and the color output from the texture stage is used as the new frame buffer pixel color.
D3DRS_SRCBLEND	A member of the D3DBLEND enumerated type.	When alpha blending is enabled this state is used to set how the source color that is about to be rendered is blended with the frame buffer. This allows us to specify an input that is used to multiply the source color and control its weight in the final color calculated.
D3DRS_DESTBLEND	A member of the D3DBLEND enumerated type.	When alpha blending is enabled this state is used to set how the source color that is about to be rendered is blended with the frame buffer. This allows us to specify an input that is used to multiply the current frame buffer color and control its weight in the final color calculated.
D3DRS_TEXTUREFACTOR	A D3DCOLOR value in the form 0xAARRGGBB. The default state is opaque white (0xFFFFFFFF)	This state can be used to set a constant color or alpha that can be accessed by the texture stage states during color and alpha blending in a texture stage. If a texture stage input argument is set to D3DTA_TFACTOR this color will be used. If the state is

RenderState	Parameters	Description
		blending two colors using the D3DTOP_BLENDFACTORALPHA color operation, the alpha component of this color is used to blend the two input colors.
D3DRS_ALPHATESTENABLE	True or False	If set to true, before a pixel is rendered its alpha value is tested against a reference value (set by the D3DRS_ALPHAREF) render state using a comparison function selected by the D3DRS_ALPHAFUNC render state. If the alpha value for a pixel fails the comparison test then it is rejected and will not be rendered.
D3DRS_ALPHAFUNC	A member of the D3DCMPFUNC enumerated type. This can be one of the following: D3DCMP_NEVER, D3DCMP_LESS, D3DCMP_EQUAL, D3DCMP_LESSEQUAL, D3DCMP_GREATER, D3DCMP_NOTEQUAL, D3DCMP_GREATEREQUAL, D3DCMP_ALWAYS. The default is D3DCMP_ALWAYS in which a pixel is never rejected based on its alpha value because it always passes the comparison test	When alpha testing is enabled this render state allows us to choose the comparison performed against the alpha reference value. For example, if we set the reference function D3DCMP_LESS with alpha testing enabled, the pixel will only pass the test and not be rejected if its alpha value is less than the reference value set by the D3DRS_ALPHAREF function. This is useful for rejecting completely transparent pixels so that they do not have their depth values written to the depth buffer.
D3DRS_ALPHAREF	DWORD	Values can range from 0x00000000 through 0x000000FF (0 to 255).

New Render States Table

RenderState	Parameters	Description
D3DRS_FOGENABLE	TRUE or FALSE	Enables fog color blending. This needs to be enabled even if you are not using the transformation pipeline and are calculating the per vertex fog factors yourself as the fog factors will still need to be interpolated and the color blending performed using these fog factors.
D3DRS_FOGCOLOR	DWORD	Enables us to set the color of the fog as an ARGB DWORD. The A component of this color is not used and can be ignored. Therefore, to set a red fog color for example, we could set the fog color to color 0xFF0000.
D3DRS_FOGVERTEXMODE	D3DFOG_NONE D3DFOG_LINEAR D3DFOG_EXP D3DFOG_EXP2	Sets the fog model used for vertex fog mode, or disables vertex fog if set to D3DFOG_NONE.
D3DRS_FOGTABLEMODE	D3DFOG_NONE D3DFOG_LINEAR D3DFOG_EXP D3DFOG_EXP2	Sets the fog model used for pixel fog mode, or disables vertex fog if set to D3DFOG_NONE.
D3DRS_FOGSTART	Float (must be passed as a DWORD)	The distance or depth at which fog color will start to be blended with our pixel or vertex when using the linear fog model. If using a vertex fog mode or pixel fog mode where 'W' based fog is being used, this should be a view space distance. If using pixel fog where 'W' based fog is NOT being used, this should be a device depth distance in the range of 0.0 to 1.0.
D3DRS_FOGEND	Float (must be passed as a DWORD)	The distance or depth at which fog color will be blended with our pixel or vertex at full intensity when using the linear fog model. If

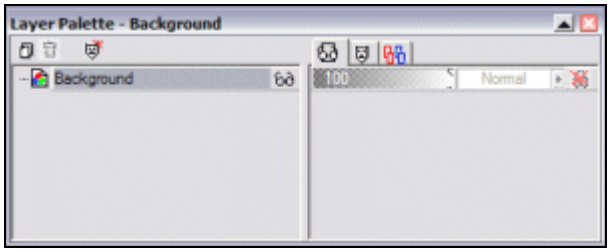
RenderState	Parameters	Description
		using vertex fog mode or pixel fog mode where 'W' based fog is being used, this should be a view space distance. If using pixel fog where 'W' based fog is NOT being used, this should be a device depth distance in the range of 0.0 to 1.0.
D3DRS_FOGDENSITY	Float (must be passed as a DWORD)	A floating point value between 0.0 and 1.0 that is used to set the fog density value for the exponential and squared exponential fog models. Not used by the linear fog model
D3DRS_FOGRANGEENABLE	TRUE OR FALSE	Available only for vertex fog mode and then only if the hardware supports 'range based' vertex fog. When enabled, the true distance from the vertex to the camera is used in the fog factor calculations eliminating rotational artifacts. If set to false, which is the default state (or if range based fog is not supported), the view space Z component of the vertex will be used instead. 'Range Based' vertex fog is more computationally expensive .

Appendix B: Creating Alpha Channels in Paint Shop Pro ™

The following is a quick guide which demonstrates creating an image which contains an alpha channel in Paint Shop Pro 7 and above (including the evaluation version).

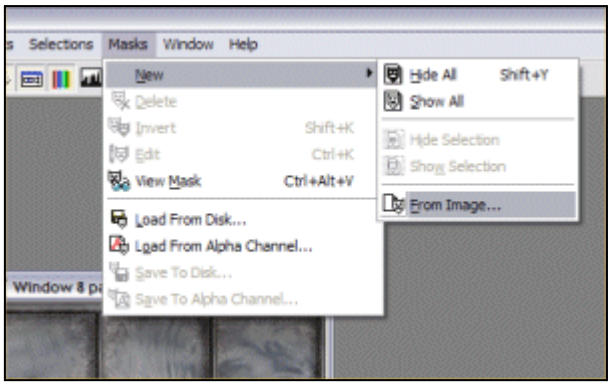
Before we begin, we need to pick an image that we wish to generate an alpha channel for. In this example we have chosen a window pane with 9 separate segments (shown to the left). We will create individual areas of translucency relatively easily by masking of the separated areas, and filling them in as needed.

After starting Paint Shop Pro you can either load the image in using traditional means (via the file / open menu), or simply drag and drop the image onto the main application work area. Once the image has loaded, we can start working on it. In this example we will not be making any adjustments to the image itself, but instead we will be working on the alpha information only. Once the image has been loaded in (assuming we simply loaded up a single layer file such as a bitmap, etc.) you can pop open the Layer Palette window, and see that we have a single Background layer, as shown below.



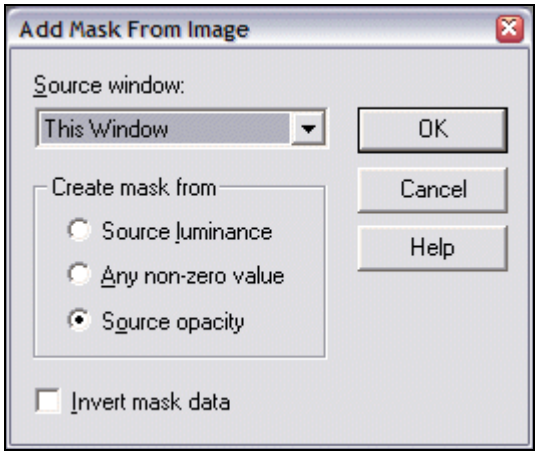
Paint Shop Pro Masks

Paint Shop Pro™ does not use alpha channels in the traditional sense. Instead, it adopts the concepts of masks. These masks can be applied to each layer individually, and can be saved out as an alpha channel in the resulting image. The first thing we need to do to apply alpha information to our image is to create a **mask**.

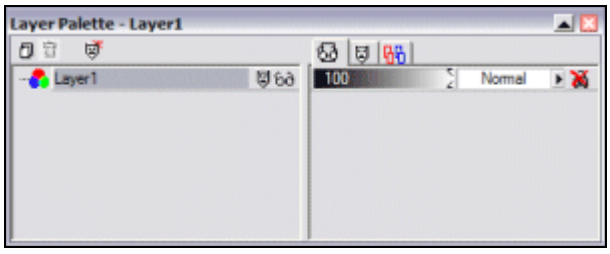


To do create a mask, we need to select the 'From Image' item from within the 'Masks / New' menu item. After selecting this item, we are presented with an options dialog as shown below. This dialog allows us to specify how we would like our alpha mask to be set up initially. For now, we just want a completely opaque mask, so we can just choose to create the mask from our source image's current opacity levels, making sure that the 'Invert Mask Data' check box is

currently unchecked. For your reference, you can use settings similar to the following to create an opaque mask from your image:



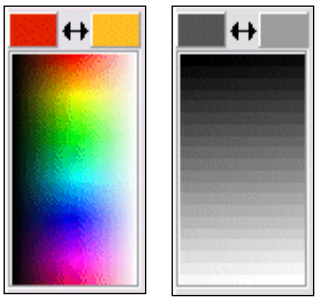
Once you have decided how your mask will be defaulted, select OK, and the mask for the currently selected layer will be generated. In this case, we had the single layer named 'Background' selected. These background layers are special types of raster layers which cannot be made translucent and are, as their name suggests, used as a background which will show through any translucent areas of any layers above it. Because of the fact that background layers cannot contain alpha information, this layer is automatically "promoted" to become a standard raster layer. It should look something like the following in the layer palette:



We can see that the icon for the layer changes to demonstrate the fact that it is no longer a background layer. In addition, its name is changed to, for example, 'Layer 1'. You can rename this layer at this point to give it a more meaningful label, but this is merely for your own benefit and plays no part in the actual process. One other important point to notice is that an additional icon has been added to the right of the layer name, which looks somewhat like a small mask. This is provided to inform you that this layer now contains a mask which can be modified, which is exactly what we will be doing next.

Editing the Layer Mask

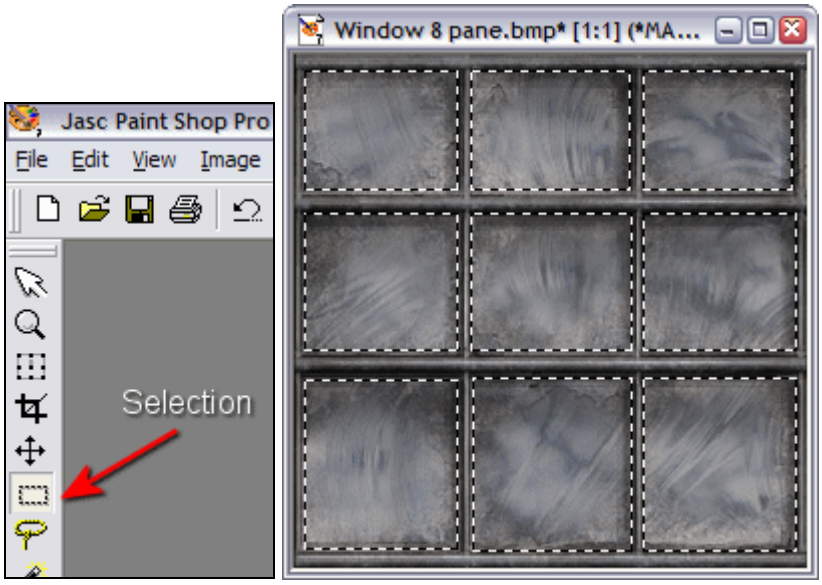
Now that we have created an empty mask, we can edit it to provide the alpha information required for our individual window panes to show through any image data rendered underneath. To do this we need to put the editor into 'Mask Edit Mode'. First of all, make sure that the layer containing the mask you want to edit is your current selection within the layer palette. Then, from the 'Mask' menu, select the 'Edit' menu item. Once you have done this, you will notice that the title bar of both the layer palette and the image itself are appended with the text '*MASK*', and that the application's color palette changes to a simple grayscale palette as shown below:



The palette shown to the left (the full color palette) is the traditional layer editing palette. The one on the right is used for editing the layer mask, and depicts the 256 levels of translucency as black (0 = Fully Translucent) through white (255 = Fully Opaque). It is worth noting however that it is often a little tricky to select the exact alpha level you want from this small quick palette. For this reason you may want to select the value from the main color palette, available by clicking in the middle of one of the 'Style' color blocks found directly underneath this quick entry palette on the 'Color' control bar.

Now we are ready to edit the mask. First we must select our alpha value, but before we do that we must make sure we are in solid color mode. To do this you can click on the black arrow contained within the 'Styles' color block underneath the palette. A small selection box will pop up allowing you to choose between 'Solid', 'Gradient', 'Pattern' or 'Null' modes. For now we want to select the solid mode as shown in the inset image. Once you are sure you are in solid mode, you can select your color from the small palette above it, or by click in the center of that same foreground style box to select the color from the main palette. We will choose a mid-range color for now, in our example palette index '151' (which has the color RGB(151, 151, 151)). By choosing an alpha value which is not totally transparent, we will be able to retain some of the original detail in each windowpane segment when it is rendered.

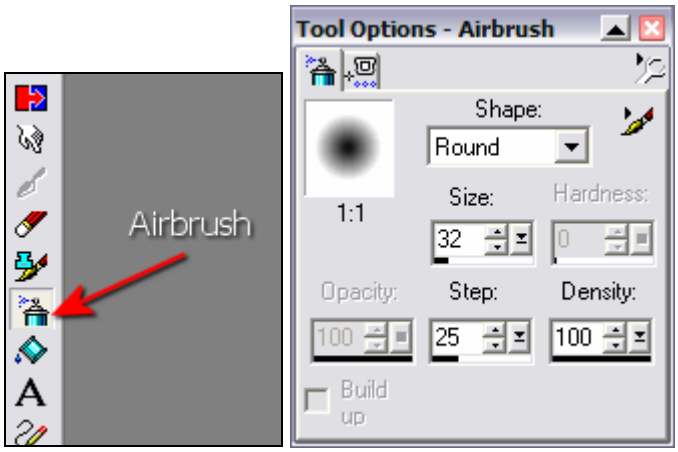
As we know, we want to leave the horizontal / vertical bars of our window pane totally opaque. This may be a little difficult to achieve, or at least a little laborious, if we were to avoid / adjust these areas by hand. We can solve this problem by using the selection tool:



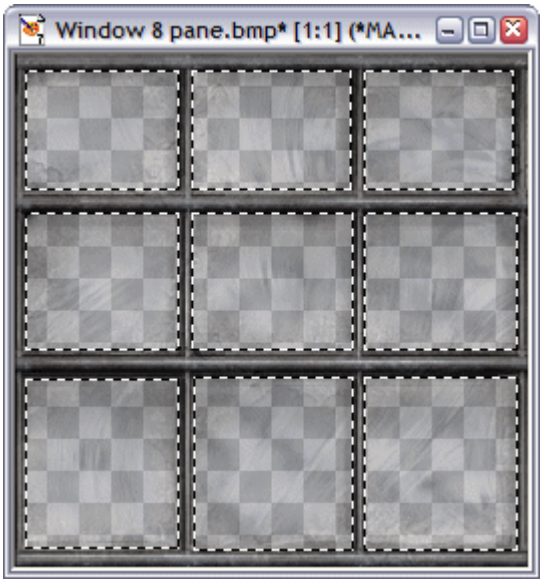
As you can see, we can use the selection tool to mask out areas of the image. Similar to many other applications of any type, you can multi-select by holding the shift key, and deselect individual areas using the ctrl key whilst in the selection mode. These are depicted by a little + or – sign being displayed next to the tool's cursor so that you can easily see which mode you are currently in.

Now that we have masked off the nine individual areas of the image, when we make any modifications, the changes will only occur inside those parts contained within the selected region(s), leaving our window pane separators completely intact. Of course, we are not currently editing the image itself, but the same applies when in mask edit mode, leaving the areas between the glass panes totally opaque.

We are now ready to modify our mask. When in mask edit mode we can treat it just as if it was a simple palletized image, and can perform many of the same color based operations with it (ex. brightness, gamma, noise, etc.) in exactly the same way. With our alpha level "color" chosen, we can now pick an editing tool. For this job, we are going to pick the airbrush with the options shown in the next image. Which tool you use for editing the alpha mask is image specific, but the airbrush suits us well for the current task.



We have chosen the airbrush, rather than simply adjusting the 'Brightness' of the alpha mask values, because this allows us to be a little inaccurate, and to go a little wild when spraying on our alpha values. This lets us leave behind dirty smudges, or shaded areas at will. Feel free to experiment with the airbrush because remember, only the areas inside your selection will be modified.



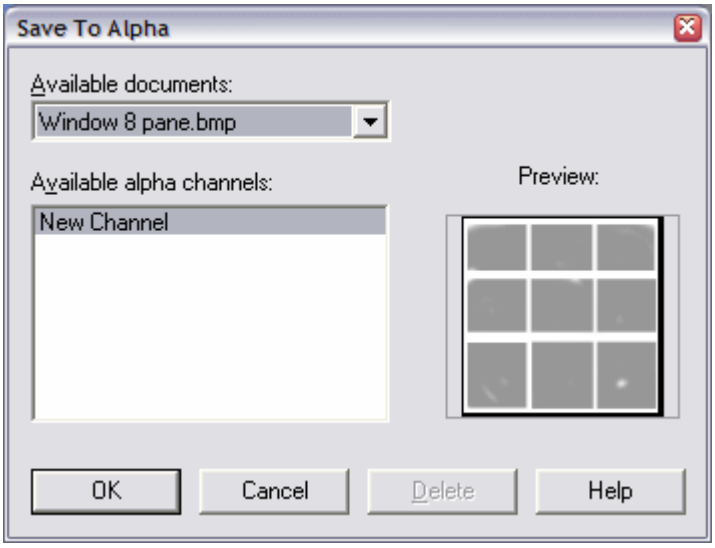
Once you are completely satisfied with the results of your haphazard airbrushing, you should end up with something a little like the image to the right, Make sure you leave your current selected areas intact. You should notice that, to help you visualize the translucent areas, paint shop has rendered a checkered pattern behind, which shows through the now translucent glass. **Tip**: This pattern can be altered on the 'Transparency' tab found via the 'File / Preferences / General Program Preferences' menu item, to allow for easier viewing in certain circumstances.

Now that we have our general alpha values set up, with our selection still intact (and still in the mask editing mode) we can touch up this image a little bit. We could for instance use the 'Noise' effect to add a little uniform noise (say 15%) which gives us a little variation in the alpha values. This can help improve the look of compressed alpha textures, or you could add texture effects to add cracks, or to allow for the distortion of rain drops.

Saving the Alpha Information

As mentioned, Paint Shop Pro™ stores its alpha information a little differently than a file would store an alpha channel, primarily because it requires per-layer alpha information. So, what we now need to do is to save our mask into the image's alpha channel. To do this we simply need to select 'Save to

Alpha Channel' from the 'Masks' menu. After selecting this item, we are presented with the following dialog:



This dialog displays a list of all the alpha channels currently stored within the image (in memory). Most file formats support only a single alpha channel, but for the moment you will not have any listed. In this case, simply select 'New Channel' from the list and press 'OK'. You will then be prompted to provide a name for this alpha channel; this is merely a description. After entering the name, the alpha mask from the selected image layer will be saved to a new alpha channel (a preview of which is displayed on the right hand side of that same dialog).

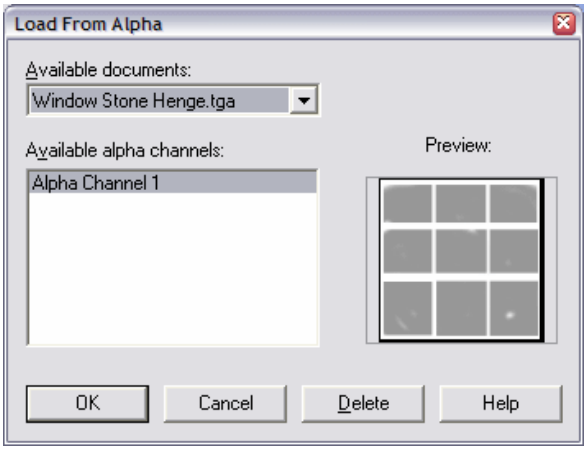
> **Note:** Alpha channels are stored separately from the masks themselves. Therefore, if at any point after saving the mask to an alpha channel, you modify that mask, you will need to save it once again to the alpha channel in the same way, overwriting the original which will be displayed in this dialog.

Before we can save our final image, it is best to save out the image as a standard .PSP file (Paint Shop Pro's™ own internal file format) so that we have a workable copy of the original image, and then to delete the alpha mask we created earlier using the 'Masks / Delete' menu item, choosing **not** to merge the mask with the selected layer. This important step needs to be performed first because otherwise, when we save to our final image format, the mask will be merged with the color layer, altering the actual color data itself. This means that if we were to render the texture, using our alpha channel, we would actually be alpha blending using the altered color data. Do not forget to remove the mask before saving as anything other than .PSP.

We are now free to save the image out to disk using the standard 'File / Save As' method, but it is important that you select a file format which is capable of storing the alpha information. Your best option here, if you are planning to load the texture and its alpha information back into Direct3D, is either .TGA or .PNG.

Loading an Image with an Existing Alpha Channel

If you want to load an image with an existing alpha channel into Paint Shop Pro, you should first load the image in the usual way. You should notice however that the alpha information is not applied to the loaded image. Remember that alpha channel information and alpha masks are separate entities. To re-create the layer mask from your alpha channel information, simply select the 'Masks / Load From Alpha Channel' menu item. You will be presented with the following dialog:



Under normal circumstances you will see only one alpha channel listed. Selecting this channel and pressing OK will result in the re-creation of the mask, and the ability to edit it once again. Remember though, that once you have edited the mask, you must save the alpha channel back out (overwriting the one in the above list) using exactly the same methods outlined in the previous section. This includes removing the mask again before you save the resulting image file.

After following these steps, you should now have a texture, available for loading into Direct3D (or any other API for that matter). Take a look at the image below which demonstrates the result of all our hard work. Stonehenge through a dirty window ☺